11

The Bash Shell

At the heart of every Mac OS X system is a Unix core. If you don't look for it, you won't see it, but it's there. Like all Unix-based systems, Mac OS X relies heavily on shell scripts. When you log in to your account at startup — whatever you happen to be doing on Mac OS X — chances are good that a shell script is involved in some way.

If you read Chapter 10, you know that scripts are simple programs that string together Unix commands to perform a task. Scripts are run by a program called a *shell*, which interprets one line of code at a time. On Mac OS X, the default shell is called Bash; Bash is a powerful shell that can be used interactively, or to run scripts.

In this chapter you learn

- How to configure and use Bash interactively and for running scripts
- How to use the Terminal application for accessing the command line
- The most important Unix commands and where to find information about commands
- Some of the commands only available on Mac OS X
- Basic shell programming

Getting Started

Before you can start interacting with the operating system via the Terminal application or writing your own shell scripts, some preliminaries need to be taken care of. First, you need to know what a *command-line interface (CLI)* is and the different ways in which it can be used. You need to have an application that can access the CLI and a way to edit scripts and other text files. Finally, you need to configure your Bash Shell before you start using it. This section covers these aspects, preparing you for the next section when you actually begin using the Bash Shell.

The Command-Line Interface

The Mac has always had an excellent graphical user interface (GUI). It's what made the Mac famous to begin with; with Mac OS X, Apple continues to lead the way when it comes to GUI design.

What was never popular under the old Mac OS was the command-line interface. A CLI is a means of interacting with the operating system via textual commands entered on the keyboard rather than by pointing and clicking the mouse. A CLI usually requires the user to enter commands at a simple prompt rather than interacting via controls and menus.

The CLI on the Mac under the old Mac OS was not popular for a number of reasons, but the most important was that earlier versions of the Mac OS did not actually have a CLI. Where Windows users could start up MS-DOS and enter commands to copy files or execute programs, Mac OS users didn't have this option; in all honesty, most didn't want it.

Mac OS X has a rich graphical interface, but it also offers a CLI as a bonus for the power user. The CLI of Mac OS X can be accessed with applications like Terminal and X11 (with `xterm`). The CLI in Mac OS X is actually the Bash Shell, which listens for the commands you enter at the prompt, and takes action accordingly.

> *X11 is the GUI used on most other Unix systems, and is equivalent to Aqua on Mac OS X. You can run X11 alongside Aqua by installing the X11 application, which is an optional install with the Mac OS X system. To install it on your Mac, either use a Mac OS X install disk or download the X11 installer package from Apple (*`www.apple.com/macosx/features/x11/`*).*

A CLI is not for everyone. Most will want to stick with what is offered in the GUI; but for others, the CLI offers an extra dimension. It is very powerful, usually offering more options to the user than can be accessed via a GUI. Some things are also much easier to do with the CLI than with a GUI. For example, Mac OS X includes Unix commands that enable you to manipulate files and text in many more ways than are possible using Finder and TextEdit.

Interactive versus Script

You can use the Bash Shell in two different ways: interactively, or for running scripts. When you type commands at a prompt, you are using the shell interactively. One command is performed at a time, as you enter it and press Return. But you can also put a series of commands in a file to form a *script*. You can use the same commands in a script as you enter at the prompt, but the script allows you to perform many commands together and execute that series of commands as many times as you please without having to retype them each time.

Working interactively does not preclude you running scripts. The two can, and usually are, interleaved. You can run any script you like from the shell prompt; usually, a *subprocess* is started to run the script, such as an extra Bash Shell. The shell initiating the script can either wait for the subprocess to exit, or continue on.

The Terminal Application

The easiest way to access Bash on Mac OS X is to use the Terminal application, which you can find in the `/Applications/Utilities` folder. Terminal can be used to open windows, each of which contains a

prompt that you can use to enter commands. Each Terminal window is running a different copy of the Bash Shell, so the commands you enter in one window do not influence the Bash Shell in another window.

When you open Terminal for the first time, you may want to change some configurations. One thing you may want to change is the default behavior of windows when the shell it is running exits. When you first use Terminal, windows remain open after a shell exits; you have to close them manually, even though they aren't useful anymore. If you want to change this behavior so that the window closes when the shell exits, choose Terminal ➪ Window Settings and select Shell from the popup in the Terminal Inspector panel that appears under When the Shell Exits. You can choose either Close the Window or Close Only if Shell Exited Cleanly. When you are finished customizing the window settings, click the Use Settings as Defaults button.

> *The Close Only if Shell Exited Cleanly option refers to the fact that each shell has an exit status when it terminates. The exit status is a simple integer number. If it is zero, the shell exited without any problem; a non-zero value indicates an error occurred. With this option, the window closes only if no errors arise.*

Apart from Terminal, you can also use the X11 application to access the command line. X11 is an optional install with Mac OS X; if you have installed it, it appears in `/Applications/Utilities`. When you start up X11, an `xterm` Terminal appears by default. `xterm` is a Unix command for starting a new terminal window in the X Windows System, which is the windowing system started by the X11 application. You can create a new terminal window in X11 either by choosing Applications ➪ Terminal or by entering the following at the prompt of an existing terminal:

```
xterm &
```

Editors

Many ways exist to edit text files on Mac OS X, including TextEdit and Xcode. Opening files in these applications from a terminal is quite easy. You can simply issue this command:

```
open filename
```

This opens the file in the Application assigned to the file type. You can choose the Application for any given file type in the Info panel of the file in Finder (select the file and then choose File ➪ Get Info).

Using external editors with Terminal is certainly possible, but may not be the most convenient solution. You may prefer to remain inside the terminal window to edit files. Unix has a vast assortment of command-line text editing programs, the most widely used of which are vi and emacs. Both are powerful editors, but they have steep learning curves and are beyond the scope of this book.

> *If you talk to Unix users about their preference for emacs or vi, you may well hit a nerve. The competition between these two editors is something akin to a religious war and can prompt very lively discussions.*

Instead of discussing vi or emacs, this chapter introduces a very simple command-line editor that ships with Mac OS X: Nano. Nano is not as advanced as emacs or vi, but it is quite adequate for basic file editing and is intuitive to use. You can edit a file with Nano simply by entering the `nano` command, followed by the filename:

```
nano filename
```

If the file already exists, it will be opened; if it does not exist, it will be created. Figure 11-1 shows a sample editing session with `nano`.

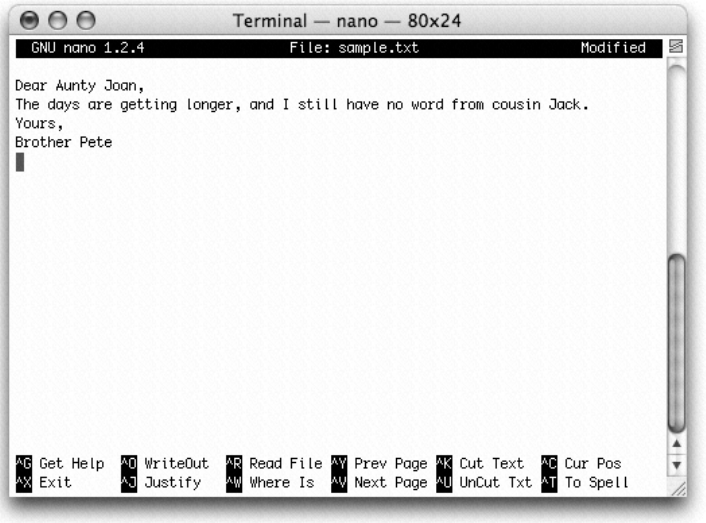

Figure 11-1

Using `nano` is relatively self-explanatory. You enter text via the keyboard and navigate with the arrow keys. Many commands also can be issued by holding down Control and pressing another key. Some of these commands appear at the bottom of the Nano editing window.

The following table provides a summary of some of Nano's more important commands. More commands are described in Help, which you can access by pressing Control-G.

Command	Description
Control-X	Exits Nano. If you have unsaved changes, Nano prompts you for whether you would like to save them.
Control-K	Cuts the selected text, or a single line if there is no selection. You can't make a selection with the mouse; instead, you set the start position of the selection with Control-^ and move the cursor to the end position with the arrow keys.
Control-U	Pastes the text that was cut with Control-K at the position of the cursor.
Control-^	Sets the starting position of a selection. After issuing this command, use the arrow keys to move the cursor to the end position of the selection.
Control-V	Moves down one full page.
Control-Y	Moves up one full page.
Control-G	Opens the help pages.

Configuring Bash

When a Bash Shell starts up, it can read a number of configuration files. You can customize your Bash Shells by entering commands in one or more of these files. It is common, for instance, to set the `PATH` *environment variable* in these files. An environment variable is a variable stored by a shell and generally influences the behavior of the shell and any programs that it executes. The `PATH` variable is a list of paths that are searched for commands and other executable programs.

When you open a Terminal window, you are greeted by an *interactive login shell*. It is *interactive* because you can interact with it in real time, and it is a *login shell* because it was not started by any other shell. When Bash starts a login shell, it executes the commands in the file `/etc/profile`. Here is the `/etc/profile` provided with Mac OS X:

```
# System-wide .profile for sh(1)

PATH="/bin:/sbin:/usr/bin:/usr/sbin"
export PATH

if [ "${BASH-no}" != "no" ]; then
        [ -r /etc/bashrc ] && . /etc/bashrc
fi
```

The first line is a comment. On the next two non-blank lines, the `PATH` variable is set, and the `export` command is used to export the variable to the shell as an environment variable. `PATH` is set to a string of colon-separated directory paths, which will be searched in order whenever a command is issued to the shell. The last three lines of the file check whether a file exists at `/etc/bashrc`; if it does, the commands in the file are executed.

> *In this chapter, the word directory is used interchangeably with folder. These words are testimony to the heritage of Mac OS X, arising out of a marriage between Unix and Mac OS. The word directory is used in the Unix world, and folder was the equivalent term in the original Mac OS. On Mac OS X, either is appropriate.*

The `/etc/profile` file is read whenever a login shell is started for any user. It should be used for system-wide configuration and not for the configuration options of a particular user. Usually, you do not need to edit `/etc/profile` at all; you can simply customize the shell from user configuration files.

After `/etc/profile`, the login shell checks whether a file called `.profile` exists in the user's home directory. If it does, the commands it includes are executed. You can put into the `.profile` file any commands you would like to run when a login shell is initiated. You could, for example, add directories to the default `PATH` variable, like so:

```
PATH=$PATH:~/bin
export PATH
```

This adds the directory `bin` in your home directory to the existing `PATH` variable. The value of the `PATH` variable is retrieved by prepending a `$` symbol, as seen on the right side of the first expression. The `export` command updates the environment variable `PATH`; without this, the variable would only be changed locally and not outside the `.profile` file.

Bash allows you to represent a user's home directory by a tilde (~) symbol. So a user's `.profile` *file is located at the path* `~/.profile`.

Not all shells are login shells. You may start one shell from within another, for example, by simply entering the command `bash` at the prompt and pressing Return. If you try this, you may not notice any change, but you are actually working inside a new shell. This type of shell is simply called an *interactive shell*; it is not a login shell.

When a new non-login interactive shell starts, the Bash Shell checks for the existence of a file called `.bashrc` in the user's home directory. If this file exists, it is executed. The `.profile` file is not executed when a non-login shell starts up. You can use the `.bashrc` file to customize your shell configuration for non-login shells. Most users don't need to have different configurations for login and non-login shells, so the `.bashrc` file often simply *sources* the `.profile` file, like this:

```
source ~/.profile
```

This command simply tells the shell to execute the commands in the file at the path `~/.profile`, which means that non-login shells are effectively configured with the same set of commands as login shells.

In the following Try It Out, you use the Terminal application with the Nano editor to create the Bash configuration files `~/.profile` and `~/.bashrc` and add a few commands to customize your Bash Shell.

Try It Out Configuring Your Bash Shell

1. Start the Terminal application in `/Applications/Utilities`.
2. Create the file `.profile` in your home directory using the Nano editor. To do this, simply issue the following command at the prompt:

```
nano .profile
```

3. Type the following text into the Nano editor. You can use the arrow keys to move around. When you are finished, press Control-X, and answer with a *Y* when prompted whether you would like to save your changes.

```
export PATH=.:$PATH:~/bin
export PS1="\h:\u:\w$ "
alias h history
```

4. Use `nano` to create the `.bashrc` file by entering this command:

```
nano .bashrc
```

5. In the Nano editor, add the following line and exit with the key combination Control-X. Be sure you save your changes to `.bashrc` when prompted.

```
source ~/.profile
```

6. When you are satisfied and want to terminate your Terminal session, enter the following command at the prompt:

```
exit
```

How It Works

This introductory example should help you get familiar with Terminal, Nano, and the Bash configuration files. When you start the Terminal application, a Bash Shell starts, and you receive a prompt. The Bash Shell always has a *current working directory*, and it begins in your home directory. When you create the `.profile` and `.bashrc` files with Nano, they are created in your home directory because that is the current working directory.

The commands added to the `.profile` file are intended to serve as examples and are not by any means compulsory. You can add whatever you like to `.profile`.

The first command extends the `PATH` variable:

```
export PATH=.:$PATH:~/bin
```

The `PATH` environment variable is initialized in `/etc/profile` before the `~/.profile` is read. The existing value is not discarded but is extended by the command. The new value of `PATH` is set to be the old value, as given by `$PATH`, with two directories added: `.` and `~/bin`. The directory represented by the period (`.`) is always the current working directory of the shell, and `~/bin` is the directory `bin` in your home directory. The directory `~/bin` does not have to exist; if it doesn't, Bash simply ignores it when searching for a command. If you create the `~/bin` directory, you could add your own scripts and other *executables* to it, and Bash would find and execute them no matter which directory you happen to be working in.

> *Any file that can be executed by the shell, whether it is a compiled program or script, is often referred to as an executable.*

Many users like to add the current working directory, as given by `.`, to their `PATH` variable. Adding `.` to the `PATH` means that the shell will look for executable programs in the current working directory, as well as at other paths. It is quite common to want to execute a program in your current directory, especially if you are writing your own scripts. If you don't include `.` in your path, you need to enter a path in order to run a script in your current working directory, like this:

```
./script_name
```

The order of the paths in the `PATH` variable is significant. The shell searches the paths in the order they appear, for the first matching executable. When an executable is found, the rest of the paths are ignored. In the example, the shell searches in the current directory (`.`) first, followed by the directories originally in the `PATH` variable, and lastly in `~/bin`. If you want the executables in a particular directory to have priority, include that directory early in your `PATH` variable.

The second line of the `.profile` file sets an environment variable:

```
export PS1="\h:\u:\w$ "
```

The `PS1` environment variable is used to formulate the prompt string that you see when the Bash Shell is waiting for you to enter a command. You can use any string you like for the prompt, as well as characters with special meanings that are substituted with another string before being displayed. In the example, the hostname (`\h`) is shown, followed by a colon and the user name (`\u`). The current working directory (`\w`) is given last, followed by a `$` symbol and a space. The following table gives some of the more interesting special characters that you can use in your prompt.

Special Character	Description
`\d`	The date (as in "Wed Nov 20th")
`\h`	The first section of the hostname
`\H`	The full hostname
`\t`	The time in 24-hour format
`\T`	The time in 12-hour format
`\A`	The time in 24-hour format, excluding seconds
`\w`	The path of the current working directory
`\W`	The last directory in the current working directory path
`\!`	The number of the command in the history list

The `.profile` file finishes by defining an *alias*. An alias in Bash is an alternative name for a command; when you type the alias, the original command to which it corresponds is executed. In this case, the `history` command, which gives a list of the commands previously given to the shell, is assigned the alias `h`. Instead of having to type `history` when you want to list the history of commands, with this alias in place you can simply type `h`.

The `.bashrc` shell, which you will recall is used to configure non-login shells, is designed in this case to simply execute the same set of commands as the `.profile` file:

```
source ~/.profile
```

This is a common approach. Using the `source` command, you can execute the contents of another file in the current shell. Note that this is not the same as running a second script, because when you use `source`, no new shell (subprocess) is started — commands are executed in the existing shell.

The `exit` command allows you to terminate a shell. You can also supply a number to the `exit` command, which is returned to the parent process as the exit code. This is usually used to indicate if an error occurred and what the error was.

Unix Basics

The Unix philosophy, which Mac OS X shares at its lower levels, can be summarized by the old adage that many hands make light work. Unix systems are full of *commands* — small programs that are highly specialized. Each command does one thing, and does it well. Even though the foundations are simple, you can achieve powerful tasks by combining Unix commands. This section covers basic aspects of Unix, some of the most important Unix commands, and how you can combine them to achieve your objectives.

Paths

Much of the time spent interacting with an operating system involves working with files and directories (that is, folders). You have to be able to locate files, view or edit them, move them, remove them, and

so forth. But all these actions require that you be able to stipulate to the operating system which file or directory a particular action involves. In Finder, you can select a file and drag it to the Trash if you want to remove it. On the command line, there are no file icons; so you need to give a path to any file or directory you want to use in a command.

Unix paths can take one of two forms: *absolute paths* and *relative paths*. Absolute paths are spelled out in full with respect to the root directory. An absolute path begins with a forward slash, as in the following:

```
cd /Users/terry/Desktop
```

This line uses the `cd` command, which changes the current working directory of the shell. The current directory is set to the `Desktop` folder of user `terry`. The path begins with a forward slash and is thus an absolute path, taken with respect to the root directory of the file system.

Relative paths do not begin with a forward slash and are taken with respect to the current working directory of the shell. If the current working directory in the preceding example is user `terry`'s home directory, the `cd` command could be issued as follows:

```
cd Desktop
```

Because the current working directory is `/Users/terry`, the home directory of user `terry`, entering a relative path of `Desktop` results in the absolute path `/Users/terry/Desktop`.

When working with relative paths, there are a few special symbols that can help you navigate. If you want to refer to the current directory, you can use a single period. The following command, for example, lists the contents of the current working directory of the shell:

```
ls .
```

The period can also be used in paths; the presence of a period effectively leaves the path unchanged. For example, the following command lists the contents of the `Desktop` folder if issued from inside the user's home directory:

```
ls ./Desktop
```

This is completely equivalent to

```
ls Desktop
```

and also to

```
ls ./././Desktop
```

The latter is nonsense, but it demonstrates the impotence of the single period in influencing paths.

Given that the single period has no effect on paths, you may be wondering why you would even need it. Sometimes it is important simply to indicate that something is a path, and a period can achieve that. For example, when issuing commands, the shell searches the paths in your `PATH` environment variable, but the current working directory is not included unless you have added it yourself. If you have an executable

in your current working directory, and you want to run it, you need to give an explicit path, otherwise the shell won't find it. Here is how you can do that:

```
./some_executable
```

Simply issuing the command without the period will result in an error message.

Another special symbol for use in paths is the double period. This moves up to the parent directory of a directory. For example, in order to list the contents of the `/Users` directory, you could enter the following from your home directory:

```
ls ..
```

Of course, the double period symbol (`..`) can also be used in paths. Here is how you could list the contents of the `/Applications` directory from your home directory, using a relative path:

```
ls ../../Applications
```

Wherever the double period occurs in the path, it moves up to the parent directory. Two double periods, as in the preceding example, effectively shift you up two levels of directories: the first one moves you to the `/Users` directory, and the second one to the root directory `/`. Once in the root directory, `Applications` selects the `/Applications` directory.

Locating and Learning Commands

Unix commands on Mac OS X tend to be stored in a number of standard directories. The most important commands appear in the `/bin` directory. `bin` stands for binary; most commands are compiled programs, which means they are in a non-readable binary format rather than a text format.

> *If you look for `/bin` in Finder, you may be surprised to find it missing. It isn't actually missing, however; it's just hidden. Apple prefers that everyday users not be bothered by low-level details like `/bin`, and hides them in Finder. You can still navigate to the `/bin` directory in Finder by choosing Go ➪ Go to Folder and entering `/bin`.*

You can list the contents of the `/bin` command by using the `ls` command. Here is the output for the command on one particular system:

```
Macintosh:~ sample$ ls /bin
bash            domainname      link            rcp             test
cat             echo            ln              rm              unlink
chmod           ed              ls              rmdir           wait4path
cp              expr            mkdir           sh              zsh
csh             hostname        mv              sleep           zsh-4.2.3
date            kill            pax             stty
dd              ksh             ps              sync
df              launchctl       pwd             tcsh
```

The `/bin` directory includes the various shells, including `bash`, as well as fundamental commands for interacting with the file system, such as `cp`, `chmod`, `mv`, and `rm`. (Details of these commands are provided throughout the "Unix Basics" section.) Even the command used to list the directory contents, `ls`, resides in `/bin`.

Mac OS X systems include a second directory intended for binaries typically used by system administrators: `/sbin`. This directory includes commands for shutting down the system and mounting volumes via a network. The commands in `/sbin` do not belong to the core of Unix commands, and many are found only on Mac OS X.

Most commands are found in the directory `/usr/bin`. This directory is intended for less fundamental commands than the ones belonging in `/bin`. Commands can be added to `/usr/bin` over time, but the contents of `/bin` are usually left intact. `/usr/bin` includes all sorts of commands, from file compression programs to compilers. Any command that is not in `/bin`, and not intended for system administrative purposes, tends to end up in `/usr/bin`. The `/usr/sbin` directory is the analog of `/usr/bin` for system administrative commands.

You can use the `which` command to get the path of a command or learn which particular path is used if there are multiple copies of a command. You simply enter `which` followed by the command name, and it prints out the path that is used if you issue the command in the shell. Here is an example of using `which` with the `emacs` command:

```
Macintosh:~ sample$ which emacs
/usr/bin/emacs
```

`which` works only with commands in the paths defined by your `PATH` environment variable. If you seek a command outside your path, you will need to use a more general file searching command like `find` or `locate`, which are described later in the chapter.

If you want to know how to use a command, or the options that it includes, you can use the `man` command. Typing in `man`, followed by a command name, opens documentation in a simple file viewer called `less`. You can navigate through the documentation by pressing the space bar and quit `less` by pressing *q*. Figure 11-2 shows the Terminal window after the command `man ls` has been issued at the prompt.

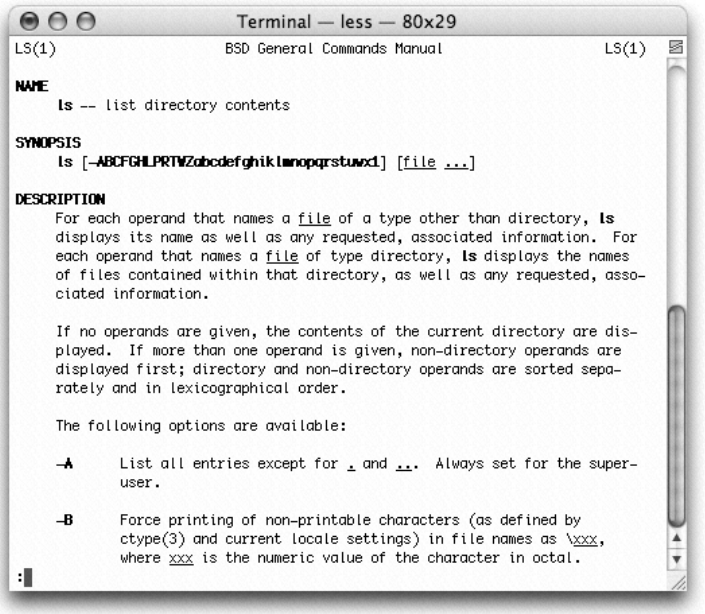

Figure 11-2

Running Commands and Other Executables

By now it should be quite clear that running an executable in a shell is simply a matter of typing its name and supplying any options required. If the command is not in one of the paths in your `PATH` environment variable, you can run the command by giving an explicit path. You can also use an explicit path if you want to override the search order used by the shell to locate a command. For example, perhaps you have two different versions of a particular command, and you want to specify explicitly which should be run. Using an explicit path to the command achieves this.

When a command or other executable is run, the shell can either wait for it to exit or continue on processing other commands. When the shell blocks and waits, the command is said to be running in the *foreground*; if the shell continues without waiting, the command is running in the *background*. By default, a command runs in the foreground. If you want to run a command in the background, you need to append an `&` symbol to the end of the line, like this:

```
find . -name "*.txt" &
```

This command searches for text files in the current directory or any of its subdirectories. Because this may take a while, it makes sense to run the command in the background so that you can continue issuing commands while the search proceeds. The `&` at the end of the line indicates to the shell that it should return from the `find` command immediately, rather than waiting for it to complete. You get a new prompt, and can continue to issue commands to the shell.

When you run a command or other executable you are actually starting up a new *process*. A process is simply a running program. Commands started from inside a shell are *subprocesses* or *child processes* of the shell and inherit the environment variables of the shell as a result. If the shell exits for some reason while the subprocess is running, the subprocess is terminated by the operating system.

You can pass arguments to a subprocess when it is started simply by including the arguments after the executable name. For example, the following `find` command is passed three arguments:

```
find / -name "*.doc"
```

The arguments are passed to `find` as the strings `/`, `-name`, and `*.doc`. The `find` command interprets the arguments and carries out the request. The shell itself has no understanding of what the arguments represent or which arguments the `find` command expects to receive; it simply breaks the line into separate strings and passes them to `find`.

When a command exits, it returns an exit value. This is an integer value, which is usually used to indicate when an error has occurred. A value of `0` means that everything went fine; a non-zero value usually indicates that an error occurred. Often the non-zero value returned is an error code, which can be used to find out exactly what type of error occurred.

To access the error code of a command in Bash, you use the special variable `$?`. `$?` is the error code of the last command run by the shell. You can test the value of `$?` after a command has run to determine if anything went wrong, comparing it to zero, for example. You learn how to perform such tests later in this chapter in the "Shell Programming" section.

If you want to exit a shell, you can use the `exit` command. With no argument, the `exit` command sets the exit code to `0`. If an error occurs, you will want to set the error code to a non-zero value. To do this, simply supply the error code as an argument to `exit`, like this:

```
exit 127
```

Here, the exit code has been set to 127.

Bash provides a few other ways to run commands and scripts. The `eval` command can be used to run commands. The commands are executed in the existing shell; no subprocess is initiated. For example, the following lists the contents of the directory `/usr/bin`:

```
eval ls /usr/bin
```

This really becomes useful only when you can evaluate arbitrary strings of commands that are read from file or entered by the user. Strings are covered in greater depth in the "Shell Programming" section.

The `source` command, which you saw earlier, is similar to `eval`, but it executes commands from a file. The commands are again executed in the existing shell, with no subprocess initiated.

With the `exec` command, you can replace the shell with another running script or program. The initiating script is terminated and the newly started executable replaces it, taking its environment, and even *process identity*, the number used to represent the process by the system. If you run the following command

```
exec ls
```

the command `ls` replaces the Bash Shell used to initiate it. When `ls` is finished listing the current working directory, it also exits. Because the shell has terminated, your prompt does not return; depending on your preferences, your Terminal window may close.

If you issue the command

```
exec bash
```

it may seem that nothing has changed, but you have actually started a new Bash Shell, replacing the old one. If you decide you want to use a different shell than `bash` during a Terminal session, you can do it like this:

```
exec tcsh
```

This replaces the existing Bash Shell with a new TCSH Shell.

Redirecting Input and Output

The real strength of shells is their ability to easily combine relatively simple commands to perform complex tasks. To achieve this, it is important to be able to take the data output by one command and use it as input to another command, or to write data to file and read it back in later for further processing. The Bash Shell provides powerful, easy-to-use features for channeling data between commands and files.

Data is channeled from one command to another, or to and from a file, via *pipes*. Pipes are analogous to the plumbing in your bathroom, except that they transmit data instead of water. To pipe data from the output of one command to the input of another, you use the hyphen (`|`) operator. Here is an example of taking the output of an `ls` command and piping it to a command called `grep`:

```
ls -1 /usr/bin | grep cc
```

Better ways exist to achieve the same effect as this command, but it does demonstrate the workings of a pipe. The command `ls -1 /usr/bin` produces a lot of output, which can be summarized as follows:

```
Macintosh:~ sample$ ls -1 /usr/bin
CFInfoPlistConverter
a2p
acid
aclocal
aclocal-1.6

...

zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
zmore
znew
zprint
```

Results differ depending on the commands you have installed in the `/usr/bin` directory. The output gets piped to the input of the command `grep cc`, which extracts any line containing the text `cc`. The original output of the `ls` command is reduced to only those commands containing the text `cc`:

```
Macintosh:~ sample$ ls -1 /usr/bin | grep cc
cc
distcc
distccd
distccschedd
gcc
gcc-3.3
gcc-4.0
perlcc
powerpc-apple-darwin8-gcc-4.0.0
rpcclient
yacc
```

You are not limited to piping data between two commands; you can pipe together as many commands as you like. By way of example, imagine that you were only interested in commands in `/usr/bin` that contained `cc` and a digit in their names. Here is one way to list those commands:

```
ls -1 /usr/bin | grep cc | grep -e '[0-9]'
```

The output of this command is

```
Macintosh:~ sample$ ls -1 /usr/bin | grep cc | grep -e '[0-9]'
gcc-3.3
gcc-4.0
powerpc-apple-darwin8-gcc-4.0.0
```

A second pipe has been added, taking the output of the `grep cc` command and piping it into the input of a second `grep`. The second `grep` prints only the lines that contain at least one digit.

You can also pipe data to and from files. To do this, you use the *redirection operators* < and >. The < operator redirects the standard input of a command causing it to be read from a file, like so:

```
grep -i TABLE < index.html
```

Here, the command `grep TABLE`, which prints any line of text containing `TABLE`, is applied to the contents of the file `index.html`. The shell reads `index.html`, channeling the data into the `grep` command, which prints those lines with `TABLE` in them.

Piping the output of a command to file is similar:

```
grep -i TABLE < index.html > table_results.txt
```

This command has been extended, with the output of the `grep` command now being piped to the file `table_results.txt`, rather than being displayed by the shell. After this command has executed, you should be able to open the file `table_results.txt` in an editor such as Nano or TextEdit and find the `grep` output there.

Notice that using > overwrites any existing file. If you want to append the data, rather than replacing the contents of the output file, you can use the >> operator:

```
grep -i TABLE < index.html >> table_results.txt
```

If `table_results.txt` doesn't exist before this command is issued, it is created and the command's output inserted. If the file does exist, the output is appended to the end of the existing data in `table_results.txt`.

Apart from standard output, every command also has a stream of data called *standard error,* which is intended for error messages. You can pipe the standard error to a file using the `2>` operator, like this:

```
grep -sdf 2> grep_error.txt
```

The `grep` option given here is invalid, so it prints an error message to the standard error and exits. The file `grep_error.txt` ends up containing the following text:

```
grep: unknown directories method
```

The form of redirection operator used here is not applicable only to the standard error but to any *file descriptor*. The standard error has the file descriptor `2`, so the operator `2>` pipes the standard error to a file. The standard output has the file descriptor `1`, so `1>` pipes data to standard output. (The standalone > operator is shorthand for `1>`.) Standard input has the file descriptor `0`; you can read from standard input with the operator `0<`, as well as with the shorthand notation <.

In the next Try It Out, you learn how to redirect data by performing a series of commands interactively in the Bash Shell. The objective is to determine the total amount of RAM available on your Mac by using the command-line tool `system_profiler`.

Try It Out Determining Your Memory by Redirecting Data

1. Open a terminal window in the Terminal application.
2. Enter the command

```
man system_profiler
```

3. Skim the information provided until you think you understand what the `system_profiler` command does. You can use the space bar to move down and press *b* to move back up a page. Press *q* when you are ready to quit the `less` viewer.
4. Back at the Bash prompt, enter the following command:

```
system_profiler SPMemoryDataType
```

The output should look something like this:

```
Macintosh:~ sample$ system_profiler SPMemoryDataType
Memory:

    DIMM0/J21:

      Size: 256 MB
      Type: SDRAM
      Speed: PC133-333

    DIMM1/J22:

      Size: 256 MB
      Type: SDRAM
      Speed: PC133-333

    DIMM2/J23:

      Size: 256 MB
      Type: SDRAM
      Speed: PC133-333

    DIMM3/J24:

      Size: Empty
      Type: Empty
      Speed: Empty
```

5. Now re-enter the command, but pipe the output to a temporary file, like this:

```
system_profiler SPMemoryDataType > sysoutput.tmp
```

6. Open the file `sysoutput.tmp` with Nano to make sure it contains this output:

```
nano sysoutput.tmp
```

7. Exit Nano again by pressing Control-X.

8. Use the `grep` command to read the `sysoutput.tmp` file, and extract the sizes of the RAM modules, like so:

```
grep -e 'Size: [0-9]' < sysoutput.tmp
```

You should see something similar to the following:

```
Macintosh:~ sample$ grep -e 'Size: [0-9]' < sysoutput.tmp
      Size: 256 MB
      Size: 256 MB
      Size: 256 MB
```

9. Repeat the command in step 8, but redirect standard output to a new file:

```
grep -e 'Size: [0-9]' < sysoutput.tmp > grepoutput.tmp
```

10. Enter the following command to extract the numbers in `grepoutput.tmp`:

```
awk '{print $2}' < grepoutput.tmp
```

You should see output that resembles this:

```
Macintosh:~ sample$ awk '{print $2}' < grepoutput.tmp
256
256
256
```

11. Repeat the command in step 10, but pipe the output to a new temporary file:

```
awk '{print $2}' < grepoutput.tmp > awkoutput.tmp
```

12. Process the `awkoutput.tmp` file with the following command:

```
perl -e '$sum=0; while(<>) { $sum+=$_; } print "$sum\n";' < awkoutput.tmp
```

The output displays the total RAM in your computer:

```
Macintosh:~ sample$ perl -e '$sum=0; while(<>) { $sum+=$_; } print "$sum\n";' <
awkoutput.tmp
768
```

Notice that the command is one long line that has been wrapped by the Bash Shell onto the next line. Do not insert a return in the command.

13. Repeat the first few commands of this chain, but instead of generating a temporary file to transfer data, just use a direct pipe from one command to the next, like this:

```
system_profiler SPMemoryDataType | grep -e 'Size: [0-9]'
```

14. Enter the following long command, to duplicate the result of steps 1 through 12 while avoiding temporary files:

```
system_profiler SPMemoryDataType |  grep -e 'Size: [0-9]' | awk '{print $2}' |
perl -e '$sum=0; while(<>) { $sum+=$_; } print "$sum\n";'
```

Again, allow the command to be wrapped by Bash; do not type a return until you have entered the whole command.

15. Remove the temporary files by entering the following three commands at the prompt:

```
rm sysoutput.tmp
rm grepoutput.tmp
rm awkoutput.tmp
```

How It Works

This example is designed to give you lots of practice piping data to and from files and between commands. The commands used throughout the example are covered later in the chapter; for now, concentrate on how data is shifted between the commands, rather than on how the commands themselves actually work.

The `system_profiler` command is used to write information about the memory in your Mac to a temporary file, using the standard output redirection operator `>`. The data in the temporary file is then read back in to the `grep` command using the standard input redirection operator `<`. This pattern is followed for the rest of the example, writing data to a file and reading it back in, with each command reducing the data a bit more until the final result is produced.

Rather than introducing temporary files that must be cleaned up later, it is often easier to pipe data directly between commands. This is the approach introduced in the last few steps of the example. Instead of writing the output of each command to a file and reading it back in to the next command, a pipe is used to channel output data from one command to the next. With this approach, the entire sequence of commands can be reduced to a single line, and no temporary files are produced.

At the end of the example, the temporary files are deleted with the command `rm`. `rm` is covered in detail later in the section "Working with Files and Directories."

Navigating the File System

Navigating the file system is somewhat different with Bash than it is with Finder. The shell maintains an environment variable, `PWD`, containing the path to the current working directory. Any relative paths you enter in your commands are interpreted with respect to this path.

Just as you can open different folders in Finder, you can also change the current working directory of a shell. The command `cd` is used for this purpose. To use `cd`, you simply pass the path to a new directory as an argument. The path can be either an absolute path or a relative path. Here is an example of using an absolute path to change to the `/Library/Frameworks` directory:

```
cd /Library/Frameworks
```

To affirm that the current directory did change, you can check the value of the `PWD` environment variable or use the `pwd` command, which prints the path of the current working directory:

```
Macintosh:/Library/Frameworks sample$ echo $PWD
/Library/Frameworks
Macintosh:/Library/Frameworks sample$ pwd
/Library/Frameworks
```

The `echo` command simply prints a string to standard output after values have been substituted for any variables by the shell.

You can also use relative paths with `cd`, in which case the path is taken relative to the current working directory. So, if the current working directory is your home directory, entering the following command will take you into your `Desktop` folder:

```
cd Desktop
```

A few special directories in your file system can be reached via shortcuts. Entering `cd` without any path will take you to your home directory. Your home directory is stored in the environment variable `HOME` and can also be represented by the tilde (~) symbol. Each of the following commands changes the current working directory to your home directory:

```
cd
cd $HOME
cd ~
```

To change to your `Desktop` directory, you could use this:

```
cd ~/Desktop
```

You can also access the home directory of another user by appending the user name to the ~. For example, to change to the `Desktop` folder of the user `terry`, you could enter this:

```
cd ~terry/Desktop
```

By default you do not have permission to change to the `Desktop` *directory of another user on Mac OS X. To be allowed to do this, the other user would have to change the permissions of the directory to give you access. The "File Permissions" section discusses this in more detail.*

Another important directory is the root directory of the file system. This is given by a single forward slash. To change to the root directory, you can issue this command:

```
cd /
```

Navigating a file system is also about knowing what you can navigate to. In Finder, you are automatically presented with a list of available files and folders whenever you open a folder. In Bash, this is not the case; you have to enter a command to list the contents of a directory. The command in question is `ls`.

If you issue the `ls` command without any arguments, it lists the contents of the current directory. In the following example, the current working directory is `/bin`:

```
Macintosh:/bin sample$ ls
bash            domainname      link            rcp             test
cat             echo            ln              rm              unlink
chmod           ed              ls              rmdir           wait4path
cp              expr            mkdir           sh              zsh
csh             hostname        mv              sleep           zsh-4.2.3
date            kill            pax             stty
dd              ksh             ps              sync
df              launchctl       pwd             tcsh
```

If you supply a path to `ls`, absolute or relative, it lists the contents of that directory, no matter what the current working directory happens to be:

```
Macintosh:/bin sample$ cd
Macintosh:~ sample$ ls /var/log/httpd/
access_log      error_log
```

The `cd` command changes the current working directory to the user's home directory. The `ls` command lists the contents of a different directory, namely the `/var/log/httpd` directory used to store log files of the Apache web server.

> *The `/var/log/httpd/` directory may be empty if you have never used your Apache web server before, in which case the `ls` command given will not print any filenames.*

The `ls` command has a number of useful options. The `-l` option allows you to get detailed information about files and directories, including their size, when they were last modified, and who owns them:

```
Macintosh:~ sample$ ls -l /var/log/httpd
total 936
-rw-r--r--  1 root  wheel  450481  5 Dec 21:35 access_log
-rw-r--r--  1 root  wheel   26433  7 Dec 19:54 error_log
```

In this example, the contents of `/var/log/httpd` have been listed again, but this time by using the `-l` option. The first part of the line indicates the *file mode*, which gives the *permissions* of each file. These determine who is allowed to read, write, or execute a given file. The meanings of the various permissions are discussed later in the chapter.

Other useful information is the file owner, which is `root` for both files in this case; the group of the file, which is `wheel` for both files; the size of the file in bytes, which is 450481 bytes for `access_log` and 26433 for `error_log`; and the date and time they were last modified.

The `-R` option is also quite useful, because it recursively lists subdirectories:

```
Macintosh:~ sample$ ls -R ~tiger/Sites
images          index.html

/Users/tiger/Sites/images:
apache_pb.gif   macosxlogo.gif  web_share.gif
```

This command lists the contents of the `Sites` directory of the user `tiger`, as well as all the subdirectories of `Sites`.

Working with Files and Directories

Knowing how to navigate the file system is one thing, but being able to modify it is just as important. The coming sections cover how you can alter the file system, copying or moving files and directories, creating them, removing them, searching for them, and even compressing and archiving them.

To move a file or directory from one path to another, you use the `mv` (move) command. But this command does more than just move a file or directory from one place to another. It can also be used to change the name of a file or directory or replace one file with another. `mv` simply changes one path, the

source path, to another path, the *destination path*; if that involves changing the name of the file or directory, that is what happens.

To begin with, consider simply moving a file from one directory to another:

```
mv somefile somedir
```

In this simple example, the file called `somefile` in the current working directory is moved into the directory called `somedir`, which is also located in the current working directory. Of course, `mv` also works with any form of relative or absolute path:

```
mv ~/Desktop/somefile .
```

In this case, `mv` moves the file `somefile` in the `Desktop` folder into the current working directory.

If a file already exists at the destination path, *it is overwritten by the moved file*. You need to be careful not to accidentally delete files you want to keep.

Changing the name of a file is no more involved. You simply ensure that the destination path either doesn't exist or is a file that you want to overwrite. In either case, `mv` moves the file to the destination path, changing its name appropriately. For example, to change the name of a file called `autumn.txt` to `spring.txt`, with both files in the current working directory, you can do this:

```
mv autumn.txt spring.txt
```

After this operation, `autumn.txt` no longer exists, and the file that used to be called `autumn.txt` is now called `spring.txt`.

Other forms of paths are also possible, of course. Here is an example where a file is moved from the user's `Desktop` folder into the `Documents` folder and renamed at the same time:

```
mv ~/Desktop/project.doc ~/Documents/lastproject.doc
```

The file originally called `project.doc` is not only moved to another directory, but its name also gets changed to `lastproject.doc`. If there is already a file called `lastproject.doc` in the `Documents` folder, it will be overwritten and lost.

If you want to avoid accidentally overwriting files when you use `mv`, you can use the `-i` option. This will cause `mv` to prompt you before it overwrites any file. To be really sure you won't accidentally overwrite a file, you can even add an alias to the `.profile` file, like this:

```
alias mv="mv -i"
```

Now, whenever you enter `mv`, it will be executed with the `-i` option included automatically.

Moving directories is similar to moving files, but there are some differences. To change the name of a directory, you simply use a destination path that does not already exist. For example, if there is a directory called `projects` in the current working directory, and you want to rename it `lastyearsprojects`, you could do this:

```
mv projects lastyearsprojects
```

Note that if there is already a directory called `lastyearsprojects`, the `projects` directory will not replace it as would happen in the case of files. Instead, the `projects` directory becomes a subdirectory of `lastyearsprojects`. If you want to replace one directory with another, you first have to either move or remove it. (Removing directories is covered shortly.)

Copying files and directories follows similar rules to moving them. The `cp` command is used to copy files from one path to another:

```
cp sourcefile destinationfile
```

Unlike `mv`, the `sourcefile` continues to exist after the `cp` operation; `destinationfile` is a duplicate of `sourcefile`. Just as with `mv`, all manner of paths can be used to stipulate the source and destination files, and if the destination file already exists, it is overwritten.

> **The `cp` command had one drawback on Mac OS X prior to version 10.4: it did not copy the resource fork of a file. The *resource fork* is metadata used by the original Mac OS to store information about a file, such as its type. Other operating systems tend to favor the use of file extensions, such as `.txt`, to delineate file type, and Mac OS X now uses a combination of file extension and other metadata.**
>
> **Mac OS X does still recognize and use resource forks to some extent; so if they exist, it is worth trying to keep them intact. On Mac OS X 10.4 and later, `cp` preserves resource forks; but on earlier versions of the system, resource forks are effectively removed by `cp`. There are, however, a few commands similar to `cp` that you can use to circumvent this problem. One is `ditto`, which can be used to copy files and directories while retaining resource forks. For more information, read the `ditto` man page by typing `man ditto`.**

To copy a directory, you have to use the `-r` option with `cp`, like so:

```
cp -r ~/Desktop/sourcedir ~/Documents
```

This copies the directory `sourcedir` in the `Desktop` folder, plus all of its contents, into a new directory called `sourcedir` in the `Documents` folder. If you wanted to rename the copied directory, you could simply do this:

```
cp -r ~/Desktop/sourcedir ~/Documents/destdir
```

Now the copy, while still located in the `Documents` folder, is called `destdir`. If the destination directory already exists, `cp` will not replace it but will make the new copy a subdirectory of the destination directory.

Both `mv` and `cp` can be used with multiple sources, as long as the destination is a directory. For example, the command

```
mv file1 file2 file3 destdir
```

moves the files `file1`, `file2`, and `file3`, which are in the current working directory, to the `destdir` directory, which is also in the current working directory. As always, any form of path can be used for the files and directories in the command.

Removing files is fairly straightforward; you simply use the `rm` command and give the path to the file:

```
rm somefile
```

This removes the file `somefile` in the current working directory. You can also remove multiple files, simply by including their paths as arguments to `rm`, like this:

```
rm ~/Desktop/temp.txt ~/rubbish.doc ~/Documents/project.txt
```

This command removes three different files, which are located in three different directories.

To remove a directory, you either have to supply the `-r` option to `rm` or use the `rmdir` command. Here is an example of each approach:

```
rm -r ~/Desktop/somedir
rmdir ~/Desktop/somedir
```

Making a new directory is achieved using the `mkdir` command. You give the path to the new directory as an argument:

```
mkdir /Users/tiger/Desktop/newdir
```

This creates a directory called `newdir` in the `Desktop` folder of the user `tiger`.

In the following example, if you tried to issue the command before first creating the `newdir` directory, an error would result:

```
mkdir /Users/tiger/Desktop/newdir/otherdir
```

`mkdir` only makes a new subdirectory of an existing directory unless you supply the `-p` option, in which case it also generates any non-existing intermediate directories. So the preceding command could be made to succeed like so:

```
mkdir -p /Users/tiger/Desktop/newdir/otherdir
```

File Permissions

All of the commands discussed so far will succeed only if you have permission to perform the requested operation. Every file and directory in the file system has a set of permissions; in Finder, you have limited access to these permissions when you select a file, choose File ➪ Get Info, and open the Ownership & Permissions section of the Get Info window. This tells you who owns the file and the operations you are allowed to perform. If you are the owner, you can also change the permissions of the file or folder.

> **This section discusses traditional Unix file permissions. Mac OS X 10.4 "Tiger" includes a second means of setting permissions for a file: Access Control Lists (ACL). Access Control Lists are considerably more flexible than traditional Unix permissions, but are also more involved. If you want to learn about ACLs, you can start by reading the man page for the `chmod` command. This command can be used to interact with ACL attributes.**

The Bash Shell gives you even more control over permissions and ownership. You can find out the permissions of a file or directory using the `ls -l` command, as explained earlier. For example, to learn the permissions of the commands in `/bin`, you could enter the following:

```
ls -l /bin
```

The output of this command lists one line for each file. Each line looks something like this:

```
-r-xr-xr-x   1 root  wheel  14380 Mar 21 00:38 cat
```

The owner or user of the file is the third entry on the line, in this case `root`. The permissions of three different types of users are given in the string at the start of the line. The first character in the string indicates the file type, with a hyphen for a file and `d` for a directory. The rest of the string can be broken into three blocks of three characters, giving the permissions of the owner, group, and other users, respectively. Figure 11-3 shows the string in detail.

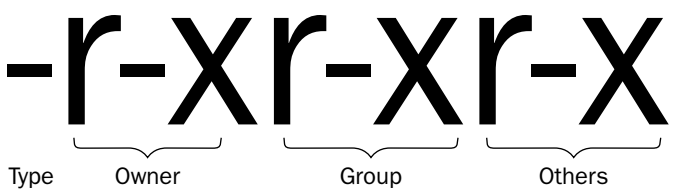

Figure 11-3

The permissions applying to the owner of the file are in places 2–4, which are `r-x` in this example. The first of these three characters indicates whether the owner has read permission, allowing the contents of the file to be examined or copied, for example. A letter `r` indicates that read permission is granted; a hyphen indicates reading is not allowed.

The second character indicates if the owner is allowed to write to the file, replacing it with another file, or changing its contents. A `w` means writing is allowed; in this example, the owner is not allowed to change the file, so a hyphen appears as the second character.

The last of the three characters pertains to whether the owner can execute the file. If a file is executable, it can be run as a program or script. An `x` here indicates that the file can be executed by the owner, and a hyphen indicates that this is not allowed. This permission is somewhat different for directories because they cannot be executed. For a directory, an `x` indicates that the owner can examine the directory's contents, listing them with the `ls` command, for example.

The remaining six characters in the string are divided between permissions for the file's group, and permissions for all other users. Each file has a group, as well as an owner. The group that a file belongs to is given as the fourth entry on the line printed by `ls -l`; in this example, it is `wheel`. Each user belongs to one or more groups; and if a given user is in the group that a file belongs to, that user has the group permissions.

If a user is not the owner of the file, and also not in the group that the file belongs to, the permissions given by the last set of three characters applies.

If you are the owner of a file, you can change its permissions. The command `chmod` is used for this purpose. `chmod` allows you to add or remove permissions for the owner, group, or other users. For example, to make a file executable by the file owner, you could do this:

```
chmod u+x ~/Desktop/somefile
```

The argument `u+x` indicates that the owner of the file, as indicated by the `u`, should be granted execute permissions for the file at the path `~/Desktop/somefile`. The plus symbol (`+`) indicates permissions should be granted, whereas a minus symbol or hyphen (`-`) would indicate permissions should be withdrawn.

Just as you can change the permissions for the file's owner by using a `u` in the argument string, you can also change permissions for the group by using `g` and other users by using `o`. You can even simultaneously update permissions for more than one type of user and/or multiple permission types. Take the following example:

```
chmod ug+rw somefile
```

This command grants read and write permission to the file owner and group users. To remove these permissions again, you can use this command:

```
chmod ug-rw somefile
```

The `root` user can change a file's owner or group. The `chown` command is used for this purpose:

```
chown terry somefile.txt
```

This changes the owner of the file given as second argument to user `terry`. The group of the file is left unchanged.

> `root` *is known as the superuser and can do anything, regardless of permissions. If you are an administrator on your Mac, you can carry out commands as if you are the superuser using the* `sudo` *command, which is discussed shortly. You need to be careful when you are carrying out commands as the* `root` *user because the results of a mistake could be drastic. A mistyped command could easily delete all the files on your file system, so be careful.*

To change the group of a file, the `root` user can issue a `chown` command like this:

```
chown :admin somefile.txt
```

The `root` user can also change the group and owner of a file, or multiple files, simultaneously with `chown`. Here is such an example:

```
chown terry:admin file1 file2 ~/Desktop/file3
```

The new owner is listed before the colon, and the group after it. The rest of the line contains the paths to the files for which ownership is to be modified.

If you are an administrator of your Mac and need to perform an operation for which you do not have permission, you can use the `sudo` command. `sudo` performs an operation as the user `root`.

By way of example, suppose you want to move a new executable to the `/usr/bin` folder. `/usr/bin` is writable only by the `root` user, so trying to move a file into it causes an error:

```
Macintosh:~ sample$ mv exefile /usr/bin
mv: rename Desktop/temp.py to /usr/bin/temp.py: Permission denied
```

To overcome this restriction, you can use `sudo`, like so:

```
sudo mv exefile /usr/bin
```

You will be prompted to enter your password; if you enter it, the `mv` command will succeed. You can use `sudo` in this way to perform any command for which you don't have permission, including `chown`.

> **The `sudo` command expects you to enter your own password when prompted, not that of the superuser `root`.**

Globbing

If you want to perform a command for several files, it can become tedious typing full names and paths. The Bash Shell allows you to take some shortcuts, using pattern matching within filenames and paths. This is known in the world of shell scripting as *globbing*.

To use globbing, you insert pattern-matching characters into strings to match certain filenames or paths. The pattern-matching characters match zero or more characters in the name or path. Probably the most widely used pattern-matching character is the wildcard `*`. This special character will match any zero or more characters. For example, used on its own, it matches all files and directories:

```
Macintosh:/bin sample$ cd /bin
Macintosh:/bin sample$ echo *
[ bash cat chmod cp csh date dd df domainname echo ed expr hostname kill ksh
launchctl link ln ls mkdir mv pax ps pwd rcp rm rmdir sh sleep stty sync tcsh test
unlink wait4path zsh zsh-4.2.3
```

This example uses the `echo` command to print out all the names that `*` matches, in the directory `/bin`. Because `*` matches any number of characters, all the files in `/bin` are echoed. The Bash Shell replaces the wildcard with the names that it matches before passing the resulting filenames as arguments to the `echo` command. The `echo` command itself is not passed the wildcard — the shell interprets any globbing characters before it runs the `echo` command.

The `*` character is also useful in combination with non-special characters. For example, suppose you want to list all the text files in your `Desktop` folder. Here is how you could do it:

```
ls ~/Desktop/*.txt
```

The path `~/Desktop/*.txt` matches any file or directory located in the `Desktop` folder with the extension `.txt`. The `*` matches any string, of any length, and `.txt` only matches names that end in exactly those characters. Together, `*.txt` matches any name ending in `.txt`.

The `*` character is also useful in the middle of a path. Suppose you want to find all `.txt` files in a subdirectory of the `Desktop` folder. Here is how you could do that:

```
ls ~/Desktop/*/*.txt
```

Notice that you can use more than one pattern-matching character in each path. The first asterisk matches any directory in the `Desktop` folder, and the second one, together with `.txt`, matches any file or directory name ending with the extension `txt`.

If you only want a wildcard that matches any single character, rather than zero or more like the `*` character, you can use `?`. `?` matches one and only one character. The following lists files that have exactly five characters in their names, with `blah` as the first four:

```
ls blah?
```

A file called `blah` will not match, but `blah0` or `blaht` will. `blah00` will not match, because `?` matches exactly one character.

If you want to limit the number of possible characters matched, you can enter the allowed characters in square braces, like this

```
ls blah[01]
```

This would match `blah0` or `blah1`, but not `blah2`, for example. You can add as many characters between the braces as you like; any of them can match.

You can use various characters to modify the behavior of the pattern-matching braces. If you insert a `^` character directly after the first brace, only characters not included between the braces will match. For example,

```
ls blah[^01]
```

will match `blah2`, but not `blah0` or `blah1`.

There are also some sets of characters that you can specify within the braces. For example, `:alpha:` represents alphabetic characters and `:upper:` represents uppercase characters. So

```
ls blah[:alpha:]
```

will match `blaht`, but not `blah0`. For a complete list of character sets, see the Bash man page.

> **Some of the need for pattern matching is removed by Bash's *autocompletion* feature. If you enter the beginning of a file or directory name at the Bash prompt, you can press the Tab key to see if Bash can complete it for you by considering the possibilities in the given context.**
>
> **If Bash can find a unique possibility, it inserts it for you, saving you some typing. If it can't, it does nothing. But if you press the Tab key a second time, it shows you all the possible matches. You can then add a few more characters to ensure a unique match and use autocompletion again to finish off.**
>
> **Autocompletion also works with commands. Try entering `system_` and pressing the Tab key. Bash should find the `system_profiler` command in your path and fill in the rest of the command for you.**

In the next Try It Out, you put theory into practice by interacting with your Mac's file system via the Bash Shell. In doing so, you use fundamental commands such as `ls`, `cd`, `mv`, `mkdir`, `chmod`, `chown`, `sudo`, and `rm`.

Try It Out Interacting with the File System

1. Open the terminal window in the Terminal application.
2. Change to the `/usr/bin` directory and list all commands that contain `gnu`.

```
Macintosh:~ sample$ cd /usr/bin
Macintosh:/usr/bin sample$ ls *gnu*
gnuattach       gnudoit         gnuserv
gnuclient       gnumake         gnutar
```

3. Copy the commands containing `gnu` to your `Desktop` folder and then change to the `Desktop` folder. After typing `Desktop` in the `cd` command, try pressing the Tab key to use Bash's auto-completion feature to finish off the path:

```
Macintosh:/usr/bin sample$ cp *gnu* ~/Desktop/
Macintosh:/usr/bin sample$ cd ~/Desktop
```

4. Make a subdirectory of the `Desktop` folder called `somecommands` and move the commands that you just copied into that subdirectory:

```
Macintosh:~/Desktop sample$ mkdir somecommands
Macintosh:~/Desktop sample$ mv *gnu* somecommands
```

5. Change to the `somecommands` directory and list its contents:

```
Macintosh:~/Desktop sample$ cd somecommands/
/Users/terry/Desktop/somecommands
Macintosh:~/Desktop/somecommands sample$ ls
gnuattach       gnudoit         gnuserv
gnuclient       gnumake         gnutar
```

6. List full details of the `gnutar` command, using `ls -l`:

```
Macintosh:~/Desktop/somecommands sample$ ls -l gnutar
-rwxr-xr-x  1 terry  terry 186024 Apr 29 20:50 gnutar
```

7. Change the permissions of `gnutar`, removing read and execute permissions for all users that are neither the file's owner nor in the file's group:

```
Macintosh:~/Desktop/somecommands sample$ chmod o-rx gnutar
```

8. Change the owner and group of the `gnutar` file to some other user on your system. In the example given here, the new user and group are `tiger`. You should use a user that exists on your system, in place of `tiger`. First try to use `chown` without the `sudo` command. An error should arise. When it does, use the `sudo` command to force the change of ownership. You will be prompted for your password.

```
Macintosh:~/Desktop/somecommands sample$ chown tiger:tiger gnutar
chown: gnutar: Operation not permitted
Macintosh:~/Desktop/somecommands sample$ sudo chown tiger:tiger gnutar
Password: **********
```

9. List full details of the `gnutar` command again. Notice how the permissions, owner, and group have changed:

```
Macintosh:~/Desktop/somecommands sample$ ls -l gnutar
-rwxr-x---  1 tiger  tiger  186024 Apr 29 20:50 gnutar
```

10. Move down one directory to the `Desktop` directory and remove the `somecommands` directory. Answer `y` when prompted as to whether you would like to override the permissions of the `gnutar` command:

```
Macintosh:~/Desktop/somecommands sample$ cd ..
Macintosh:~/Desktop sample$ rm -r somecommands
override rwxr-x---  tiger/tiger for somecommands/gnutar? y
```

How It Works

Most of this example involves fairly straightforward changes of directory and copying and moving of files. These are very common actions when working in a shell, and you should learn them well.

In order to list the files containing `gnu` in the `/usr/bin` directory, two special globbing characters were used:

```
Macintosh:/usr/bin sample$ ls *gnu*
```

Because the `*` character matches zero or more characters, `*gnu*` matches any file or directory name containing `gnu`, including cases where `gnu` is at the beginning or end of the name.

After copying the files containing `gnu` to the `Desktop` folder, a new directory is created with `mkdir`, and the commands moved into it with `mv`. The `gnutar` command is then singled out for practicing modification of ownership and permissions. First, the rights of other users to read or execute the file are removed, using `chmod` with the argument `o-rx`. `o` refers to users that are neither owner nor in the file's group. The hyphen means that rights are being revoked, and the `rx` indicates that the rights being revoked are for reading and executing. (Other users did not have write permissions to begin with, so they do not need to be revoked.)

Next, an attempt is made to change the owner and group of the `gnutar` file. This does not succeed at first because normal users are not allowed to change ownership of a file; only `root` can change ownership. To overcome this impedance, the `sudo` command is used to run the `chown` command.

To finish off the example, the `somecommands` directory is deleted. Because you do not own the `gnutar` file, you are prompted if you would like to override the permissions. If you answer in the affirmative, the `gnutar` file and the rest of the `somecommands` directory are deleted.

Searching for Files

With the coming of Mac OS X "Tiger," and the Spotlight search technology it contains, you have some pretty powerful tools for finding files and directories available to you on Mac OS X. In addition to the graphical interface, Apple has provided tools for searching with Spotlight from the command line. You can use Spotlight to search for files by name, but it also searches file content and metadata.

Metadata is data about data. It includes information such as a file's name, the date it was created, the type of data it contains, and much more. If you want to know what metadata is associated with a particular file, you can use the `mdls` command to find out:

```
Macintosh:~ sample$ mdls Sites/index.html
Sites/index.html -------------
kMDItemAttributeChangeDate     = 2005-04-21 19:26:03 +0200
kMDItemContentCreationDate     = 2003-10-28 11:05:45 +0100
kMDItemContentModificationDate = 2003-10-28 11:05:45 +0100
kMDItemContentType             = "public.html"
kMDItemContentTypeTree         = ("public.html", "public.text", "public.data",
"public.item", "public.content")
kMDItemDisplayName             = "index.html"
kMDItemFSContentChangeDate     = 2003-10-28 11:05:45 +0100
kMDItemFSCreationDate          = 2003-10-28 11:05:45 +0100
kMDItemFSCreatorCode           = 0
kMDItemFSFinderFlags           = 0
kMDItemFSInvisible             = 0
kMDItemFSLabel                 = 0
kMDItemFSName                  = "index.html"
kMDItemFSNodeCount             = 0
kMDItemFSOwnerGroupID          = 501
kMDItemFSOwnerUserID           = 501
kMDItemFSSize                  = 5754
kMDItemFSTypeCode              = 0
kMDItemID                      = 485368
kMDItemKind                    = "HTML document"
kMDItemLastUsedDate            = 2003-10-28 11:05:45 +0100
kMDItemTitle                   = "Mac OS X Personal Web Sharing"
kMDItemUsedDates               = (2003-10-28 11:05:45 +0100)
```

Spotlight command-line tools are located in the `/usr/bin` directory and begin with the letters `md`, which stand for *metadata*. As you can see from the output of the `mdls` command, even a simple HTML file has a lot of metadata associated with it. Each *metadata attribute* is associated with a key; the keys can be seen in the left column of the output of `mdls`, with the data value itself given in the right column.

To search the metadata and content of files, you can use the `mdfind` command. For example, to find all files that include the text `Personal Web Sharing` in the metadata or content, you could issue the following command at the prompt:

```
mdfind "Personal Web Sharing"
```

You can restrict your search to particular metadata attributes by using a simple query string. For example, to search only the metadata attribute `kMDItemFSName` of each file, which contains the filename, you could issue the following command:

```
mdfind "kMDItemFSName == 'Personal Web Sharing'"
```

The string passed to `mdfind` is a query string; in this case the path to any file whose `kMDItemFSName` attribute includes `Personal Web Sharing` will be written out.

Spotlight is an exciting new technology, but it does have one drawback — it works only on Mac OS X 10.4 and later. If you are working on an earlier version of Mac OS X, or a completely different Unix-based

operating system, Spotlight will not be available. Luckily, there are some traditional Unix tools that can help you locate files and directories by name.

One of the easiest way to find paths containing a given string is to use the `locate` command. For example, to find all paths containing the string `NSApplication.h`, you could issue the following command:

```
Macintosh:~ sample$ locate NSApplication.h
/Developer/ADC Reference
Library/documentation/Cocoa/Reference/ApplicationKit/Java/Classes/NSApplication.htm
l
/Developer/ADC Reference
Library/documentation/Cocoa/Reference/ApplicationKit/ObjC_classic/Classes/NSApplica
tion.html
/System/Library/Frameworks/AppKit.framework/Versions/C/Headers/NSApplication.h
```

As you can see, even a reasonably unique string like this can generate several results. Bear in mind that the string can match anywhere in the path. If it matches a directory name, for example, the whole contents of the directory, and all of its subdirectories, will be printed. Try to be specific about what you are searching for when using `locate`.

The `locate` command works with a database of all the files on your file system, which gets updated weekly by Mac OS X. This means that new files are unlikely to be found by the `locate` command. `locate` is useful for finding files that do not change often, such as commands, libraries, and header files. It will not be very effective for finding regularly changing files and directories, such as those in your own projects.

> **The `locate` database gets updated once a week on Mac OS X, but only after it has been created. You usually have to create the initial database yourself. To do this, issue the command `sudo /usr/libexec/locate.updatedb` in Terminal.**

A command you can use to search the file system in its current state is `find`. Of course, `find` doesn't have the benefit of a database, so it is slower than `locate`. It has to go through the file system one file/directory at a time, checking the filename, and reporting results. But `find` has many more options than `locate`, and is a powerful command to learn.

The most common way of using `find` is like this:

```
find /System/Library/Frameworks -name NSApplication.h
```

This command searches in the directory `/System/Library/Frameworks`, and all subdirectories, for any file or directory with the name `NSApplication.h`.

You can also use globbing characters in the name you pass to `find`. To search for any file with a name containing `darwin` in the current working directory or any subdirectory, you could enter this:

```
find . -name "*darwin*"
```

The `*` characters match zero or more characters in the name, so the path of any file whose name contains `darwin`, including those that begin or end with `darwin`, will be printed.

You can also carry out commands on the files you locate with `find`. For this, you use the `-exec` option, which runs a command that you provide, like this:

```
find . -name rubbish -exec rm {} \;
```

This command finds files called `rubbish` in the current working directory or any of its subdirectories. When a file called `rubbish` is found, the command given after `-exec` is performed. The command in this case is `rm`, to remove the file. The path of the located file can be accessed using the special character `{}`, so `rm {}` is the same as typing `rm` followed by the path to the file. Commands following the `-exec` option need to be terminated by a semicolon, but because the semicolon has a special meaning to the Bash Shell, it is escaped with a backslash.

The `find` command is very powerful, with many options. For example, you can list or carry out commands on files that were last modified before a given date. Perhaps you want to remove them after a while. In fact, your Mac OS X system uses the `find` command every day to clean up old files. Take a look in the file `/etc/daily`, which is a Bash script run by Mac OS X every day. You should be able to find a section like this:

```
if [ -d /tmp ]; then
    cd /tmp && {
    find . -fstype local -type f -atime +3 -ctime +3 -exec rm -f -- {} \;
    find -d . -fstype local ! -name . -type d -mtime +1 -exec rmdir -- {} \; \
        >/dev/null 2>&1; }
fi
```

In short, this complicated set of commands uses `find` to remove temporary files and directories located in the `/tmp` directory, after they have not been accessed for a number of days. For more information on the options used in these commands, and `find` in general, read the `find` man page:

```
man find
```

Working with Text

An important part of using a shell is being able to manipulate text, whether the text is the contents of a file or the output of a command that needs to be piped into a second command. Unix systems have a number of powerful tools for working with text, including `grep`, `sed`, and `awk`. This section covers the most important commands for handling text on Mac OS X.

A simple but commonly used command is `echo`. `echo` simply prints out whatever follows it on the line, after the shell has substituted any variable values or special characters. For example, to print a greeting to standard output, you could do this:

```
Macintosh:~ sample$ echo Hello Cupertino !
Hello Cupertino !
```

But you can make your message a bit more flexible by including variables and other special Bash characters:

```
Macintosh:~/Desktop sample$ touch Cupertino
Macintosh:~/Desktop sample$ GREETING=Hello
Macintosh:~/Desktop sample$ echo $GREETING Cuper* !
Hello Cupertino !
```

The `touch` command simply updates the time stamp of the file given as an argument; or, if the file doesn't exist, it creates a new empty file with that name. In this case, an empty file called `Cupertino` is created.

A variable called `GREETING` is then initialized to `Hello`, and the `echo` statement combines the value of the variable, as given by `$GREETING`, with the names of all files and directories in the current directory beginning with `Cuper`. `Cuper*` is interpreted by the shell as a glob and expanded before the string is passed to `echo`. The net result is precisely the same as the first example of using `echo`.

The `cat` command can be used to concatenate (join) a number of files, printing the result to standard output. If used with a single file, it simply writes the contents of the file to standard output. To join two files, creating a third, you could do this:

```
cat file1 file2 > file3
```

After this command, `file3` will first contain the contents of `file1`, followed by the contents of `file2`. `file1` and `file2` themselves will be unaltered. If `file3` exists prior to the operation, it will be overwritten.

`cat` is often used to initiate a chain of operations on the contents of a file rather than using the < redirection operator. For example, you could extract all lines of a file containing the word `Cupertino` by issuing the following command:

```
cat somefile | grep Cupertino
```

The `cat` command writes the contents of `somefile` to standard output. The standard output is piped into a `grep` command, which prints any line containing the text `Cupertino`.

`grep` is a very useful command, which can be used to extract lines of text matching some pattern. In the simplest case, it just searches for a literal match to a string:

```
Macintosh:~/Desktop sample$ grep '<TABLE' index.html
<TABLE WIDTH="85%" BORDER="0" CELLSPACING="15" CELLPADDING="0">
```

The `grep` command searches the file `index.html` for any lines that include `'<TABLE'`, which is an opening tag for a HTML table. Notice the use of single quotes; this is necessary because the < character has a special meaning to the shell (it is the input redirection operator). If you enclose a string in single quotes, the shell will ignore any special meanings, and pass the string unaltered to `grep`.

`grep` can also be used with special pattern-matching strings called *regular expressions*. You can think of regular expressions as being similar to globbing, but do not confuse the two. Although the characters used for pattern matching in globs and regular expressions are often the same, their meanings can be quite different.

Here is a simple example of using a regular expression with `grep`:

```
Macintosh:~/Desktop sample$ grep 'BORDER.*CELLSPACING' index.html
<TABLE WIDTH="85%" BORDER="0" CELLSPACING="15" CELLPADDING="0">
```

The pattern to be matched is `'BORDER.*CELLSPACING'`. This regular expression matches any line that includes the text `BORDER`, followed by zero or more arbitrary characters, and then the text `CELLSPACING`. The period (`.`) character is special in regular expressions. It matches exactly one character, like the `?` in

Bash globbing. The * character has a very different implication in regular expressions; it means that the preceding character can occur zero or more times. In this case, the period is the preceding character, so it can be used to match zero or more arbitrary characters. The complete pattern thus matches any string beginning with CELLSPACING and ending with BORDER, with any characters in between.

Two types of regular expression that you can use with grep are *basic* and *extended*. There is not enough room to cover all the intricacies of regular expressions in this chapter, but they are powerful and well worth learning. To get you started, here is a table of the most important extended regular expression patterns, and what they match. To use extended regular expressions, insert the -E option in front of the regular expression. Basic regular expressions are similar to extended regular expressions, but more restrictive. See the man page of grep for more details.

Pattern	Description	Example	Does Match	Doesn't Match
.	Matches any single character.	char.t	char0t chartt	chart
*	Modifies the meaning of the previous character. A match can occur with zero or more of the previous character in the regular expression.	char*t	chat chart charrt	
+	Similar to *, but requires at least one match. Zero occurrences does not match.	char+t charrrt	chart charrt	chat
{n}	Matches exactly *n* occurrences of the previous character.	char{3}blah	charrrblah	chablah charblah charrblah
[...]	Matches any character given in the square braces.	char[xYz]t	charxt charYt charzt	chart charyt charwt
(...)	Used to group characters together.	char(xyz)*blah	charblah charxyzblah charxyzxyzblah	charxblah charxyblah
|	Allows two different expressions to match.	char(x|yz)+blah	charxblah charyzblah charxxblah charxyzblah charyzxblah charyzyzblah	charblah charzyxblah charyyzzblah

grep is a good command for extracting lines of text, but it can't help you much if you want to modify text. For that you need a more powerful tool, such as sed or awk.

awk is named after its creators — Aho, Weinberger, and Kernighan — and is actually a whole language unto itself. Unfortunately, there isn't the space here to do it justice. It can be used for complex text

manipulation, but you can also use it for simple tasks like extracting a word of text based on its position in a line. For example, suppose you wanted to use the `wc` command to count the number of words in a file. You could do something like this:

```
Macintosh:~/Desktop sample$ wc index.html
     125     799    5754 index.html
```

`wc` provides the number of lines, words, and characters in the given file. But you are interested only in the number of words; you can extract this number with the following command involving `awk`:

```
Macintosh:~/Desktop sample$ wc index.html | awk '{ print $2; }'
799
```

In this case, the output of `wc` has been piped to `awk`. An `awk` program is given in between single quotes. `awk` programs process one line of text at a time, in a repetitive manner. This very simple `awk` program prints the second whitespace-separated entry on each line that it reads. Because only one line is output by the `wc` command in this example, `awk` prints only the string `799`, which appears second on the line output by `wc`.

> *The number of words in a file can be obtained more easily by simply supplying the* `-l` *option to* `wc`*. However, this would not have demonstrated the usefulness of* `awk`*, which was the purpose of the preceding example.*

The `sed` command is not a complete programming language like `awk`, but it is a very powerful editor. A common use for `sed` is to replace one regular expression with another. For example, the following command replaces any occurrences of the text `Apple`, with `Apple Computer Company`:

```
sed -e 's/Apple/Apple Computer Company/g' somefile.txt
```

The resulting text is written to standard output. The first argument to `sed`, which is enclosed in single quotations, is an editing command; it tells `sed` what text substitution to make. A substitution command takes the form `s/.../.../g`, with the regular expression between the first and second forward slashes replaced by the text between the second and third slashes.

Here is a more advanced example, leveraging the possibilities of regular expressions:

```
sed 's/< *[Tt][Aa][Bb][Ll][Ee].*>/<table class="tableclass">/g' index.html
```

This example scans the file `index.html` for any HTML opening tags for tables. These tags take the basic form `<TABLE . . .>` and can include attributes after the `TABLE` label. The `sed` command searches for these tags and replaces them with `<table class="tableclass">` when found.

The regular expression used to find the `TABLE` tags is quite involved. It looks like this:

```
< *[Tt][Aa][Bb][Ll][Ee].*>
```

It begins with the `<` character, which is followed by a space and a `*` character. The `*` indicates that a match can include zero or more of the previous character, namely, the space. A series of five square braces follow, each containing an upper- and lowercase letter. Together, these braces account for the case-insensitivity of HTML, allowing for every legal form of the string `table`, including `TABLE`, `table`,

TaBlE, and tabLE. The regular expression handles the possibility of attributes by including the characters .*, which match zero or more arbitrary characters after the table label. The > terminates the tag.

In the following Try It Out, you use some of the commands you have learned for manipulating text to change the table cell widths in an HTML file.

Try It Out Editing an HTML File with sed

1. Open a window in the Terminal application.
2. At the prompt, change to your Desktop directory using the cd command:

```
Macintosh:~ sample$ cd ~/Desktop
/Users/tiger/Desktop
```

3. Copy the default index.html file from your Sites folder to the current working directory:

```
Macintosh:~/Desktop sample$ cp ~/Sites/index.html .
```

4. Use cat to pipe the contents of the index.html file into the input of grep, extracting any lines that include tags of the form <TD ...>:

```
Macintosh:~/Desktop sample$ cat index.html | grep -i '<TD.*>'
                <TD COLSPAN=3>
                <TD WIDTH=50%>
                <TD WIDTH=5%></TD>
                <TD WIDTH=45%>
```

5. Using sed, replace all these tags with the tag <TD WIDTH=100>, and put the resulting HTML in a file called new.html:

```
Macintosh:~/Desktop sample$ sed 's/<TD.*>/<TD WIDTH=100>/g' index.html > new.html
```

6. Confirm that the tags have been substituted by using on the file new.html the same grep command as in step 4:

```
Macintosh:~/Desktop sample$ grep -i '<TD.*>' new.html
                <TD WIDTH=100>
                <TD WIDTH=100>
                <TD WIDTH=100>
                <TD WIDTH=100>
```

How It Works

If you haven't modified your Sites directory since installing Mac OS X, it should still include a default index.html file, which is used in this example. If you don't have this file anymore, you can use any HTML file you like as long as it has TD tags.

In step 4, grep is used to extract any line containing an opening table cell tag, which has the label TD. The -i option stands for *case-insensitive*, so grep ignores case, effectively matching tags containing the labels TD, tD, Td, and td.

The sed command replaces the opening table cell tags with the tag <TD WIDTH=100>, using a substitution. Results are redirected with the > shell operator to the file new.html. Notice that in this case, no

effort has been made to treat the possibility of tag labels including lowercase letters. This could be achieved by using the pattern `<[Tt][Dd].*>`, but it is not necessary here because the `grep` command in step 4 already shows that there are no lowercase tags.

The last `grep` command confirms that the tags have been updated as expected. You can also examine the `new.html` file for yourself with Nano if you want to be sure everything went as planned.

Process Control

Running short subprocesses in the foreground is fairly straightforward, but if you have something more time-consuming, you will want to run it in the background. You need to be able to monitor the progress of your background process, to find out its status, and whether it has exited. Even if you don't initiate background processes very often yourself, many are run by the `root` user to handle all sorts of administrative activities.

> **Many background processes are *daemons*. Daemons run continuously, and are often started when the system boots up. They typically listen for requests on a network port and act on them when they are received, perhaps starting up other programs.**
>
> **If you want to see what sort of daemons are running on your system, open the Activity Monitor utility. In an Activity Monitor window, select Administrator Processes from the Show popup button in the toolbar.**

The `ps` command is used to check the status of processes running on your system. If you use it without any arguments, it prints the processes associated with the shell you are using:

```
Macintosh:~ sample$ ps
  PID  TT  STAT      TIME COMMAND
 1112 std  S      0:00.14 -bash
```

Only one process is shown in this case: the Bash Shell itself. Other information given includes the `PID`, or process identifier; `STAT`, the status of the process; and `TIME`, the CPU time consumed. The `PID` is needed to identify the process in any commands that you issue. The possible status values are given in the following table.

Status Value	Description
D	The process is in disk, and cannot be interrupted.
I	The process is idle. It has been sleeping for longer than 20 seconds.
R	The process is running.
S	The process has just been put to sleep. It has been sleeping for less than 20 seconds.
T	The process is stopped.
Z	The process is a zombie. It is dead.

When you start a process in the background, it appears in the list produced by `ps`. For example, the following runs the `sleep` command in the background:

```
Macintosh:~ sample$ sleep 10 &
[1] 1140
Macintosh:~ sample$ ps
  PID  TT  STAT      TIME COMMAND
 1112 std  S      0:00.18 -bash
 1140 std  S      0:00.01 sleep 10
```

The `sleep` command simply sleeps for the time you give as an argument, in this case 10 seconds. The `&` has been used to put the command into the background. When you do this, the `PID` of the subprocess is printed, which is `1140` in this example. When the `ps` command is issued, the `sleep` process shows up with the expected `PID` of `1140`.

Without options, `ps` supplies you only with information about the processes that you own and that were started from the shell. You can get information about other processes that you own using the `-x` option. The output of `ps -x` includes any applications that are running, whether you started them yourself or not. For example, apart from the applications you initiated yourself, there are processes for the Finder, Dock, and iCal, which you may not have realized existed.

`ps` can also be used to get information about other processes running on the system. Many options are available, and you should take a look at the `ps` man page to decide which ones you would like to include. Using `-aux` with the command provides enough information for most purposes:

```
Macintosh:~ sample$ ps -aux
USER     PID %CPU %MEM      VSZ    RSS  TT  STAT STARTED      TIME COMMAND
tiger     469  22.9  1.1    92000   7312  ??  S    Sat08PM   3:40.46
/Applications/Utilities/Terminal.app/Contents/Mac
tiger    1108   7.9  9.4   370816  61356  ??  S     5:27AM   7:29.47
/Applications/Microsoft Office 2004/Microsoft Wor
root    1158   3.0  0.1    41932    352 std  R+    6:16AM   0:00.01 ps -aux
tiger     179   3.0  4.8    83264  31700  ??  Ss   Sat06PM  17:17.38
/System/Library/Frameworks/ApplicationServices.fr
tiger    1115   1.0  1.3   103288   8716  ??  S     5:32AM   2:19.98
/Applications/Utilities/Activity Monitor.app/Cont
tiger     527   0.6  4.1   134160  26616  ??  S     7:59AM  18:00.30
/Applications/Internet Explorer.app/Contents/MacO
root    1116   0.3  0.1    27736    576  ??  Ss    5:32AM   1:14.86
/Applications/Utilities/Activity Monitor.app/Cont
root     120   0.0  0.1    27480    348  ??  Ss   Sat06PM   0:03.21 netinfod -s
local
root     122   0.0  0.0    18056    120  ??  Ss   Sat06PM   0:20.49 update
...
```

The output has been truncated, but you can see that this set of options gives you information about all processes, and for all users.

One time you will need `ps` is when something goes wrong and you want to terminate a process. If you can find out the process identifier using `ps`, you can issue a `kill` command to stop it. To demonstrate, the `sleep` command is utilized again:

```
Macintosh:~ sample$ sleep 3600 &
[1] 1180
Macintosh:~ sample$ ps
  PID  TT  STAT      TIME COMMAND
 1112 std  S      0:00.23 -bash
 1180 std  S      0:00.01 sleep 3600
Macintosh:~ sample$ kill -9 1180
Macintosh:~ sample$ ps
  PID  TT  STAT      TIME COMMAND
 1112 std  S      0:00.23 -bash
```

The `sleep` process is set to sleep for 3600 seconds, or one hour. Instead of waiting for it to exit by itself, the `kill` command is used with the option `-9` and the `PID`, in order to terminate the process. The last `ps` command confirms that the `sleep` process has exited.

> **You can use `kill` to terminate a process in the background, but what about when the process is in the foreground? Simply type Control-C, and the foreground process will be killed.**

You can also kill processes by name, rather than process identifier. The `killall` command is for this purpose.

```
Macintosh:~ sample$ sleep 3600 &
[1] 1183
Macintosh:~ sample$ killall sleep
[1]+  Terminated              sleep 3600
```

`killall` kills any process belonging to you that matches the name you pass. Processes of other users, including `root`, are unaffected.

If you do need to kill a process belonging to the root user, or some other user, you can use the `sudo` command, along with either `kill` or `killall`. `sudo` performs the command as `root`, and can thus terminate any process. Be careful not to terminate vital processes on your system or you could crash your Mac.

One drawback of `ps` is that it is static. It prints information about processes running at the time you issue the command, but it never updates. The `top` command gives similar information to `ps`, but it continuously updates. Simply issue

```
top
```

and press *q* when you are ready to exit. `top` gives all sorts of information, too much to cover here. It is particularly useful though for monitoring the percentage of CPU time that each process is using. Perhaps your system is responding slowly, and you want to know if there is a process hogging the CPU. `top` can tell you. If you use the `-u` option, it even orders the processes based on the percentage of CPU time they are using so that you can quickly find the culprit at the head of the list.

Mac OS X Exclusive Commands

If you have experience with other Unix systems, most of the commands discussed in this chapter will be familiar to you. But Mac OS X has a number of commands that are non-standard, and yet very useful.

There are too many to cover in detail, but the following table should give you an idea of what's available. By referencing the man pages, you should be able to figure out how to use each one. If a man page is not available, most commands provide help when you supply the `-h` option.

Command	Purpose
`diskutil`	Command-line equivalent of the Disk Utility application. Can be used to check and repair disks and permissions, and erase and partition disks.
`ditto`	Used primarily for copying files and directories while maintaining file resource forks. It can also be used to create compressed ZIP archives with resource forks intact.
`hdiutil`	Creates, mounts, and manipulates disk images.
`installer`	The command-line interface to the Installer application; used to install applications that come in packages.
`mount_afp`	Mounts an AppleShare file system. This is akin to choosing Go ➪ Connect to Server in Finder.
`open`	Opens any document or application, just like you would by double-clicking an icon in Finder.
`pmset`	Used to control power management. This is of interest to laptop users, because `pmset` offers more control than is available through the System Preferences application.
`system_profiler`	Provides the same information found in the System Profiler utility.
`SystemStarter`	Used to start and stop system services from the command line, such as the Apache web server.

Overview of Other Commands

It is impossible to cover all Unix commands in a single chapter, so a selection of the most important has been presented here. But there are many other commands that you may have use for. The following table gives some of the standard Unix commands that have not been discussed in detail so that you can decide for yourself which are useful. Use the man pages to learn more.

Command	Purpose
`crontab`	Used to schedule commands to run at regular times. For example, you could schedule a regular cleanup of temporary files, or a backup of your home directory.
`curl`	Command-line tool for downloading and uploading files. Supports FTP and HTTP protocols.
`dig`	Tool for interacting with domain name servers (DNS). Can be used to determine the IP address of a host, for example.

Command	Purpose
`ftp`	Used for transferring files to and from servers using the FTP protocol.
`gzip/gunzip`	Compresses/decompresses files with the GZIP format.
`netstat`	Used to get detailed information about the network that the computer is attached to.
`nice/renice`	Used to set or change the priority of a running process so that the kernel allots more or less CPU time to it.
`sftp`	Secure version of `ftp`, based on the SSH protocol. Use it to securely transfer files to or from a server.
`ssh`	Secure shell for remotely accessing a system. It can be used to log in to a remote system, giving access to a command-line interface. It can also be used to run commands on a remote system, and even allows you to encrypt network traffic between two computers using a technique known as *tunneling*. Tunneling can also be used to avoid limitations imposed by a firewall.
`tar`	Creates and extracts archives of files and directories. An archive is a single file that can contain the data of many files and/or directories.
`zip/unzip`	Compresses/decompresses ZIP archives.

Shell Programming

You can do a lot with Unix commands, but if you have to issue the same commands over and over, it can become pretty tedious. Luckily, it's possible to combine a sequence of commands into a script, which can be run as often as you like. This section shows you how to write and run scripts.

Bash also offers a number of rudimentary programming constructions, such as conditional branching and loops, which may be familiar to you from other programming languages. Although these aspects of Bash can be used when working interactively, they are most useful when writing shell scripts. The programming aspects of Bash are also covered in this section.

Scripts

Bash scripts are made up of nothing more or less than the commands you enter at the prompt when working interactively. If you enter these commands in a text file, one per line, in the order they are issued, and you change the permissions of the file so that it can be executed, you have a Bash Shell script.

Take this interactive session:

```
Macintosh:~ sample$ cd ~/Desktop/
/Users/tiger/Desktop
Macintosh:~/Desktop sample$ touch blah
Macintosh:~/Desktop sample$ mkdir blahdir
Macintosh:~/Desktop sample$ mv blah blahdir/
```

If you find yourself repeating these commands often, you might consider inserting them into a script. The script would look like this:

```
#!/bin/bash
cd ~/Desktop/
touch blah
mkdir blahdir
mv blah blahdir
```

The script simply contains the same commands that were entered at the interactive prompt. The only difference is the first line, which is called a *shebang*. A shebang begins with the characters `#!`, and ends with a path. It tells the shell that is running the script which program to use to interpret it. In this case, another Bash Shell is being used, so the path to the `bash` command is given. If this were a Perl script, for example, the path to the `perl` command would be given in the shebang.

When you have added this script to a file, you can run it in two ways. The first is to explicitly use the `bash` command with the file as an argument:

```
bash scriptfile.sh
```

The `bash` command is used to run the script in `scriptfile.sh`. If you take this approach, you do not need to have execute permissions for the file `scriptfile.sh`. Also, the shebang will be ignored because you are explicitly passing the script as an argument to `bash` rather than letting the shell decide what to run the script with.

The second and more common way to run a script is to give the script file execute permissions, like this:

```
chmod u+x scriptfile.sh
```

With the script now executable, you can run it like this:

```
./scriptfile.sh
```

You need to include an explicit path to the file, unless you have the current working directory in your `PATH` environment variable.

When you issue this command, the shell you are using examines the shebang and starts a new Bash Shell that interprets the script.

Variables

You are already acquainted with environment variables in Bash, but not all variables are environment variables. A variable becomes an environment variable when the `export` command is used. If a variable is not exported, it is visible within the script in which it is defined, but not, for example, in any subprocesses.

Variables are defined by simply assigning them. They do not have to be declared beforehand, as in some other languages. Here is a simple example of defining and using a variable:

```
ADDRESS='1 Shell Street'
echo $ADDRESS
ADDRESS='2 Shell Street'
echo $ADDRESS
```

Running this script results in the following output:

```
1 Shell Street
2 Shell Street
```

The `ADDRESS` variable is initially assigned to the string `'1 Shell Street'`. The quotation marks are important; without them, the `ADDRESS` variable would be assigned only to `1`, and the shell would not know how to interpret the rest of the string, resulting in an error. Later in the script, the `ADDRESS` variable is reassigned to `'2 Shell Street'`.

To access the value of a variable, you prepend a `$` symbol. To summarize, when assigning a value to a variable, you use the variable name without a `$`, and when you want to substitute the value of a variable, you do use the `$`. In cases where the shell cannot determine the variable name, you can use curly braces to clarify matters. Take this script, for example:

```
SUBJECT=care
echo $SUBJECTless
```

This results in an error because the shell looks for a variable called `SUBJECTless`, which doesn't exist. Curly braces are used to fix the problem:

```
SUBJECT=care
echo ${SUBJECT}less
```

The script will now print the text `careless`.

You will have noticed by now that variable names are usually written in capital letters. This is purely a convention; it is not compulsory. You can use lowercase letters or mix upper- and lowercase; however, if you want your scripts to be easily read by others, consider sticking to the convention.

In each of the preceding examples, the shell substitutes the value of any variables *before* the `echo` commands are called. This applies to all commands, not only `echo`. Unless you take certain steps to explicitly prevent variable substitution (discussed in the next section, "Quoting"), the shell substitutes any variables before running the command.

The Bash Shell also provides array type variables. Arrays allow you to include multiple values in a single variable. You access the stored values using an index. Here is an example of defining and using an array:

```
ADDRESS[0]=Hello
ADDRESS[1]=there
ADDRESS[10]=Bob!
echo ${ADDRESS[0]} ${ADDRESS[1]} ${ADDRESS[2]} ${ADDRESS[10]}
```

This script prints this output:

```
Hello there Bob!
```

You index an array using an integer in square braces, with indexes beginning at 0. As you can see, it is not necessary to assign a value for all indexes. In this example, values have been assigned only for indexes 0, 1, and 10. In the `echo` statement, the value for index 2 is requested. No value has been assigned for index 2, but it does not result in an error; instead, the value is simply an empty string. Only the values of the other three array entries actually appear in the printed output.

When you access the value of an array element, you have to use the curly braces, as shown in the example. If instead you write something like this:

```
echo $ADDRESS[1]
```

the shell first tries to retrieve the value of `$ADDRESS`. In Bash, this evaluates to the first element in the array, `${ADDRESS[0]}`. This value will be substituted and combined with `[1]`, resulting in the output `Hello[1]`, which is not what was intended.

Variables in Bash are global; that is, they are visible throughout the whole script after they have been defined. If you want to limit the visibility of a variable, you can use the `local` keyword. The following example shows a function that defines two variables, one global and one local:

```
VarFunc() {
  VAR1=value1
  local VAR2=value2
}
VarFunc
echo VAR1 is $VAR1
echo VAR2 is $VAR2
```

Here is the output of this script:

```
VAR1 is value1
VAR2 is
```

Without going into the semantics of functions, which are discussed a little later in this chapter, it should be clear that `VAR1` is visible outside the function `VarFunc`, and `VAR2` is not, as witnessed by the fact that an empty string is printed in the second line of the output, rather than the text `value2`.

You can perform arithmetic with integer variables using the `let` command. You simply write the expression you want to evaluate on the line after `let`, like this:

```
Macintosh:~ sample$ ONE=1
Macintosh:~ sample$ let THREE=$ONE+2
Macintosh:~ sample$ echo $THREE
3
```

You cannot perform arithmetic with decimal (floating-point) numbers directly in Bash. If you want to do this, you need to use the command `bc` *or a more powerful language such as Perl or Python (see Chapter 10).*

Apart from the variables that you define yourself, the Bash Shell defines a number of useful variables. For example, the process identifier of the shell running the script is given by `$$`. You can use this when naming files to avoid overwriting output from other runs:

```
echo It\'s a good day, la la la la la > goodday_output_$$.txt
```

This one-line script writes a string to a file. The filename includes the process identifier (`goodday_output_5006.txt`) so it is unlikely to overwrite another output file produced by the same script because it will have a different process identifier.

Another set of important shell variables are the *positional parameters*, which correspond to the arguments passed when the script is run. The path of the script is passed in the variable `$0`, and the arguments are passed in the positional parameters `$1`, `$2`, `$3`, and so forth. The variable `$#` gives the number of positional parameters. To illustrate, suppose that the following script is inserted in the file `argscript.sh`:

```
#!/bin/bash
echo The script is called $0
echo There are $# positional parameters
echo First argument is $1
echo Second argument is $2
echo Third argument is $3
```

When this script is run, like this:

```
./argscript.sh arg1 arg2 arg3
```

the following output is produced:

```
The script is called ./argscript.sh
There are 3 positional parameters
First argument is arg1
Second argument is arg2
Third argument is arg3
```

All arguments can also be found in the variable `$@`. Inserting the following line in the preceding script

```
echo The arguments are $@
```

results in this additional line of output:

```
The arguments are arg1 arg2 arg3
```

Another commonly used shell variable is `$?`. This gives the exit status of the last command executed in the foreground. Usually, a value of 0 indicates the command succeeded and a non-zero value indicates failure. Here is an example:

```
mkdir tempdir$$
echo Exit code of mkdir was $?
mkdir tempdir$$
echo Exit code of mkdir was $?
```

The second `mkdir` command causes an error because the directory already exists. The exit code of the successful command is 0, and that of the unsuccessful operation is 1, as you can see from the script output:

```
Exit code of mkdir was 0
mkdir: tempdir5224: File exists
Exit code of mkdir was 1
```

Quoting

Several different types of quotation marks are used in shell programming, and it is important to know the implications of each. For example, double quotations do not have the same meaning as single quotation marks, and the two are often not interchangeable.

Double quotation marks are used to form strings, as you might expect. The shell will perform variable substitution within double quotations, as you can see from the following interactive session:

```
Macintosh:~/Desktop sample$ NAME=David
Macintosh:~/Desktop sample$ TIME="6 o'clock"
Macintosh:~/Desktop sample$ MESSAGE="Meet $NAME at $TIME"
Macintosh:~/Desktop sample$ echo $MESSAGE
Meet David at 6 o'clock
```

The definition of the `TIME` variable shows that single quotes have no special meaning in a double-quoted string. When defining the `MESSAGE` variable, the values of the `NAME` and `TIME` variables are substituted before the string is assigned to the `MESSAGE` variable.

If you want to avoid the special meaning of `$NAME` and `$TIME` in the shell, you can use single quotation marks:

```
Macintosh:~/Desktop sample$ NAME=David
Macintosh:~/Desktop sample$ TIME="6 o'clock"
Macintosh:~/Desktop sample$ MESSAGE='Meet "$NAME" at $TIME'
Macintosh:~/Desktop sample$ echo $MESSAGE
Meet "$NAME" at $TIME
```

Just as you can use single quotes in a double-quoted string, you can also use double quotes in a single-quoted string. The variable substitutions made in the preceding double-quoted string are not made when single quotes are used.

If you want to use double quotation marks, but force the shell to treat certain characters literally, even when they have a special meaning, you can use the backslash, like so:

```
Macintosh:~/Desktop sample$ NAME=David
Macintosh:~/Desktop sample$ TIME="6 o'clock"
Macintosh:~/Desktop sample$ MESSAGE="Meet \$NAME at $TIME"
Macintosh:~/Desktop sample$ echo $MESSAGE
Meet $NAME at 6 o'clock
```

A backslash preceding a character escapes any special meaning that that character has. It applies not only to variables, as demonstrated here, but to any characters that have special meaning to the shell.

Another type of quotation mark used often in shell programming is the *backtick*. When a command is enclosed in backticks, the command is executed, with the output replacing the command itself. Here is how you could use this approach to rewrite the first example above:

```
Macintosh:~/Desktop sample$ NAME=David
Macintosh:~/Desktop sample$ TIME="6 o'clock"
Macintosh:~/Desktop sample$ MESSAGE="Meet `echo $NAME` at `echo $TIME`"
Macintosh:~/Desktop sample$ echo $MESSAGE
Meet David at 6 o'clock
```

In this case, the `echo` commands given in the definition of `MESSAGE` are carried out by the shell and substituted before the string is assigned to `MESSAGE`. Backticks can be very useful when you want to store the results of a command for further processing. For example, you could list the contents of a directory, storing the resulting string in a variable, as demonstrated by this command:

```
DESKTOP_CONTENTS=`ls ~/Desktop`
```

The `ls` command lists the contents of the user's `Desktop` folder and assigns the resulting output to the `DESKTOP_CONTENTS` variable.

Conditional Branching

Most programming languages provide a mechanism for branching based on a condition or the outcome of a test. This is called *conditional branching*, and Bash also supports it with the `if` command.

The most difficult aspect of learning to use `if` in Bash is constructing conditions. Here is how you could use `if` to test whether an error occurred during the execution of a `mkdir` command:

```
if mkdir hellodir
then
    echo Making hellodir succeeded
else
    echo Making hellodir failed
fi
```

The `if` statement tests if the command given as the condition has an exit value of 0, which indicates success. If so, the commands after the `then` command are executed. If the exit value of the command is non-zero, the commands after the `else` command are executed. The `fi` keyword is used to close the `if` statement.

> *In Bash, the closing keyword of a command is often just the command written backwards. For example, for* `if` *it is* `fi`*, and for* `case` *it is* `esac`*.*

This `if` command may seem confusing at first because languages such as C and Java behave in the opposite manner: the `if` block is executed when the condition is non-zero or true, and the `else` block is executed when the condition is zero or false. This difference comes about because Bash treats an exit value of zero as success, and all other values as errors.

There is an operator in Bash that allows you to get behavior more similar to what you find in other languages: the `((...))` operator. The preceding example can be rewritten like this:

```
mkdir hellodir
if (( $? ))
then
    echo Making hellodir failed
else
    echo Making hellodir succeeded
fi
```

In this example, the exit value of the `mkdir` command, which is given by the shell variable `$?`, is tested with the `((...))` operator. The `((...))` is an arithmetic evaluation operator, which is actually equivalent to the `let` command. It evaluates the expression, returning 0 if the expression evaluates to a non-zero value, and 1 if the expression is 0.

If rather than performing arithmetic operations, you want to compare string values, you can use the `[[...]]` operator. Here is an example in which the values of three variables are compared using various operators:

```
VAR1="some string"
VAR2="$VAR1"
VAR3="other string"

if [[ $VAR1 == $VAR2 ]]; then
    echo VAR1 and VAR2 are the same
else
    echo VAR1 and VAR2 are different
fi

if [[ $VAR1 != $VAR3 ]]; then
    echo VAR1 and VAR3 are different
else
    echo VAR1 and VAR3 are the same
fi
```

Here is the output of this script:

```
VAR1 and VAR2 are the same
VAR1 and VAR3 are different
```

First, notice that the `then` command, which was previously included on the line following the `if`, is on the same line in this example. This is made possible by the addition of a semicolon behind the `if` command. Separate shell commands can either be written on separate lines or can appear on the same line separated by semicolons. Using the semicolon with an `if` command makes the code a bit more compact.

The assignment of the variables to various strings at the beginning is fairly straightforward, except for `VAR2`. You may be wondering why double quotation marks have been used around the value `$VAR1` on the right-hand side of the assignment. If you don't do this, the shell will expand `$VAR1` as the two words `some` and `string`. The quotation marks used to assign `VAR1` are *not* part of the variable; they are simply there to group the separate words into a single string. If you don't use quotation marks when assigning `VAR2`, it gets assigned to the first word, `some`, and the shell won't know how to treat the extra word `string`. It's a good idea to get into the habit of using double quotes when accessing the value of any string variable, including paths, which often contain spaces.

The conditional expressions themselves are comparisons between strings. The `[[...]]` operator returns 0 if the expression is true, and 1 otherwise. The operators `==` and `!=` can be used in the string comparisons and test for equality and inequality of the strings, respectively. You can also use logical operators, as demonstrated by this example, which extends the preceding script:

```
if [[ $VAR1 == $VAR2 && $VAR2 == $VAR3 ]]; then
    echo VAR1, VAR2, and VAR3 are all equal
elif [[ $VAR1 == $VAR2 && !($VAR2 == $VAR3) ]]; then
    echo VAR1 and VAR2 are equal, but VAR2 and VAR3 are not
elif [[ $VAR1 != $VAR2 && $VAR2 == $VAR3 ]]; then
    echo VAR1 and VAR2 are not equal, but VAR2 and VAR3 are
else
    echo VAR1 and VAR2 are not equal, and neither are VAR2 and VAR3
fi
```

Here is the output for this section of the script:

```
VAR1 and VAR2 are equal, but VAR2 and VAR3 are not
```

This example introduces the `elif` command, which stands for *else if*. If the condition of an `if` command is not met, control moves to the first `elif`. If the condition of the `elif` is met, such that the expression evaluates to 0, the commands after the next `then` command are evaluated, and then control jumps down to `fi`. If the `elif` condition is not met, control moves to the next `elif`, and so on. If none of the `elif` conditions are met, the commands after `else` are executed.

The conditional expressions in this example make use of many of the available operators. `&&` is the logical AND, which evaluates to true if the expressions on either side of it are true. `||`, which is not used here, is logical OR, which is true if either expression is true. Parentheses, such as those used in the first `elif` expression, can be used to group terms. The unary `!` operator is the NOT operator and negates the value of the expression, changing true to false, and vice versa.

The final way of writing test conditions is with the `test` command, which is equivalent to the `[...]` operator. `test` can be used for arithmetic comparisons, but it is most useful for testing file attributes. Here is an example that tests if a particular file exists:

```
if test -e ~/Desktop/tempfile
then
    echo tempfile exists
fi
```

This can also be written as follows:

```
if [ -e ~/Desktop/tempfile ]
then
    echo tempfile exists
fi
```

The `test` command takes an option used to determine the type of test to be performed for the file given. In this case, the `-e` indicates that the existence of the file is being tested. If the file exists, the `test` command evaluates to 0 and the `if` block is performed, echoing the fact.

You can perform many other tests with the `test` command by using various options. The following table shows some of the more useful ones.

Option	Test Description
`-a` or `-e`	Tests if a file exists. A directory is also considered a file, for this purpose.
`-d`	Tests if a file exists and is a directory.
`-f`	Tests if a file exists and is a regular file, not a directory.
`-r`	Tests if a file exists and is readable.
`-w`	Tests if a file exists and is writable.
`-x`	Tests if a file exists and is executable.

Table continued on following page

Option	Test Description
`-O`	Tests if a file exists and is owned by the user.
`-G`	Tests if a file exists and is owned by a group of the user.
`-nt`	Binary operator used to determine if one file is newer than another. The comparison is based on the last modification dates of the two files. The test is true if the file to the left of the operator is newer than the file to the right.
`-ot`	Similar to `-nt`, but tests if one file is older than another.

The Bash Shell includes a second conditional branching command that will not be demonstrated here: `case`. A `case` command can be used to test an expression for equality with a number of possible values. For more information on the `case` command, see the `bash` man page.

Looping

Loops allow you to perform a series of commands repetitively, without having to duplicate them. The `while` loop continues until the exit status of a command is no longer zero. For example, here is a `while` loop that has a fixed number of iterations:

```
let i=0
while (( $i < 5 ))
do
    echo $i
    let i=$i+1
done
```

Here is the output of this script:

```
0
1
2
3
4
```

The command given after the `while` keyword is executed, and its exit value is checked. If it is zero, indicating success, the commands between `do` and `done` are executed. This repeats until a non-zero exit value is encountered.

In the example, the command utilizes arithmetic evaluation to determine if the variable `i` is less than `5`. If so, the loop continues; if not, it exits. The commands inside the `while` block echo the variable and then increment it by one, using the `let` command.

> *The Bash Shell also includes an* `until` *loop, which has the same structure as the* `while` *loop. The difference between the two is that the* `until` *loop continues as long as the conditional command returns a non-zero exit status and stops when an exit status of zero is encountered.*

`while` loops are often used to iterate over command-line arguments, using the `shift` command. Assume the following script is in a file called `script.sh`:

```
#!/bin/bash

while (( $# ))
do
    echo $1
    shift
done
```

When run with this command:

```
./script.sh 1 2 3 4
```

the output is

```
1
2
3
4
```

In this example, the `while` tests how many input arguments are left, which is given by the shell variable `$#`. When there are none, the `while` loop exits. Each iteration, the command `shift` removes one input parameter — the one in the `$1` variable — and moves all the others to a lower index. `$2` becomes `$1`, `$3` becomes `$2`, and so forth.

The `for` loop is capable of performing the same operations as the example shown at the beginning of this section, with somewhat less code:

```
for (( i=0 ; i<5 ; i++ ))
do
    echo $i
done
```

This is very similar to the `for` loop in the C language, which is described in Chapter 6. Three arithmetic expressions are given inside the arithmetic evaluation parentheses `((...))`. The first is evaluated just once, at the beginning. This is usually used for initializing a counting variable, like `i`. The next is evaluated every iteration, including the first. If it is true, the loop continues; if not true, the loop exits with control transferred immediately to the `done` statement. If the second statement is true, the third statement is evaluated and usually increments the counter variable; the `++` operator increases a variable by one. Each iteration, the commands between `do` and `done` are executed.

Another form of the `for` loop can be used to iterate over a list of words. In the example that follows, it is used to echo a number of names from a string variable:

```
NAMES="Bill Bob Barry Bernice Beatrix"
for name in $NAMES
do
    echo $name
done
```

This is the output:

```
Bill
Bob
Barry
Bernice
Beatrix
```

The `for ... in ...` form of the `for` loop sets the variable given after the `for` keyword to each of the whitespace-separated words in the string given after `in`.

A useful variation on this loop involves leaving out the `in` part of the loop. When you do this, the `for` loop iterates over the input arguments, as demonstrated by this script:

```
#!/bin/bash

for name
do
    echo $name
done
```

Inserting this in a file called `script.sh`, and running the script like this:

```
./script.sh Bill Bob Barry Bernice Beatrix
```

leads to this output:

```
Bill
Bob
Barry
Bernice
Beatrix
```

Functions

If your scripts start to become large, or you find yourself duplicating a lot of code, you may consider using functions. Functions allow you to group commands together, so that they can be executed from anywhere in a script with a *function call*.

You write a function like this:

```
function ChangeToDesktopAndList() {
    cd ~/Desktop
    ls
}
```

This function is called `ChangeToDesktopAndList`. The `function` keyword is optional, but the parentheses following the function name, and the braces, are required. To call this function, you simply enter the function name, like this:

```
ChangeToDesktopAndList
```

The function definition must precede the function call in the script so the shell knows about the function's existence. Wherever you call the `ChangeToDesktopAndList` function, the commands in the function are executed.

Like scripts, functions can also take arguments. And also just like scripts, arguments are accessed in a function using the shell variables `$1`, `$2`, `$3`, and so forth. Here is a more general function than `ChangeToDesktopAndList`:

```
function ChangeToDirAndList() {
    cd "$1"
    ls $2
}

ChangeToDirAndList ~/Desktop -l
```

This function changes to a directory passed as the first argument, and lists the contents, with the options of the `ls` command passed as the second argument. The function call simply lists the arguments separated by whitespace, just as if it were a script. In the example, the `Desktop` folder is the first argument, and the `-l` option is the second.

Note that if your arguments include spaces, you need to use quotation marks to group them. Quotation marks have also been used in the function around the `$1` argument, just in case the path passed includes spaces. Here is a function call in which the arguments include spaces:

```
ChangeToDirAndList "$HOME/Desktop/some dir" "-l -a"
```

The only complication is that the environment variable `HOME` has had to be used in place of the ~ because the shell does not substitute for the ~ in a quoted string.

If you need to explicitly return from a function to the calling code, you can use the `return` command. For example, in this function:

```
function ReturningFunc() {
    ls ~/Desktop
    return
    ls ~
}
```

the last `ls` command is never performed because the function jumps from the `return` command back to the calling code.

In the next Try It Out, you use the various shell programming constructions you have learned to develop a script to find and compress any large files in a given directory and any of its subdirectories.

Try It Out — Writing a Shell Script to Compress Large Files

1. Open a terminal window in the Terminal application.
2. Change to the `Desktop` directory by issuing the following command:

```
cd Desktop
```

3. Create a new file with Nano called `compress.sh`:

```
nano compress.sh
```

4. Enter the following script with Nano in the `compress.sh` file:

```
#!/bin/bash

# Check exit status in $?.
# If non-zero, print it, along with the message in $1.
# If $? is non-zero, exit script if $2 is equal to "YES".
function CheckExitStatus() {
    local EXIT_STATUS=$?
    if (( $EXIT_STATUS )); then
        echo An error occurred, with status $?
        echo Message: $1
        if [[ $2 == "YES" ]]; then
            exit $EXIT_STATUS
        fi
    fi
}

# Function that finds any files larger than 5Mb, and compresses them.
# The directory path $1, and subdirectories, are searched for large files.
function CompressFilesInDirTree() {
    find "$1" -size +5000000c -type f -exec gzip "{}" \;
    CheckExitStatus "Find command failed" YES
}

# Main program. Loop over directories passed via command line arguments.
# Compress any large files in each directory tree.
for dirPath
do
    CompressFilesInDirTree "$dirPath"
done
```

5. Change the permissions of the `compress.sh` file so that it is executable:

```
chmod u+x compress.sh
```

6. Locate a directory tree with a variety of files smaller and larger than 5MB. Copy the whole directory to a temporary directory in the `Desktop` folder:

```
cp -r source_directory ~/Desktop/temp_dir
```

`source_directory` should be the path to the directory you want to copy.

7. Run the `compress.sh` script to compress the large files in `temp_dir`.

```
./compress.sh temp_dir
```

8. Check that the files larger than 5MB that reside in `temp_dir` and its subdirectories have been compressed by the `gzip` command and have the `.gz` extension.

9. Remove the `temp_dir` directory when you are ready.

```
rm -r temp_dir
```

How It Works

The `compress.sh` script begins with a function that checks the exit status of the last command executed and acts based on whether an error occurred:

```
# Check exit status in $?.
# If non-zero, print it, along with the message in $1.
# If $? is non-zero, exit script if $2 is equal to "YES".
function CheckExitStatus() {
    local EXIT_STATUS=$?
    if (( $EXIT_STATUS )); then
        echo An error occurred, with status $?
        echo Message: $1
        if [[ $2 == "YES" ]]; then
            exit $EXIT_STATUS
        fi
    fi
}
```

Before the function definition, there is a comment. A comment can be created by simply using the `#` symbol; anything after the `#` on the line is ignored by the shell.

The function name is `CheckExitStatus`, and the first executable line defines a variable called `EXIT_STATUS`, which is visible only inside the function because of the presence of the `local` keyword. `EXIT_STATUS` is assigned to the last exit status, which is given by `$?`. Because `$?` will be reset every time a command is issued, its value is saved in the `EXIT_STATUS` variable.

The `if` block checks if the exit status is non-zero, in which case a couple of messages are echoed. The first argument to the function is a message to the user that is printed if a command failed. The second argument is used in the nested `if` command; if the argument is equal to the string `YES`, the script exits.

The second function does the bulk of the work, searching for files in a particular directory tree and compressing the large ones:

```
# Function that finds any files larger than 5Mb, and compresses them.
# The directory path $1, and subdirectories, are searched for large files.
function CompressFilesInDirTree() {
    find "$1" -size +5000000c -type f -exec gzip "{}" \;
    CheckExitStatus "Find command failed" YES
}
```

The `find` command performs the search. It has a number of options passed to it. The first argument to the function is also used as the first argument to the `find` command. This argument is the root directory of the search. (Note that double quotes have been used to ensure that any spaces in the directory path are not misinterpreted.)

The first option passed to `find` is `-size +5000000c`. This means that `find` should ignore any file smaller than 5000000 bytes, or 5MB. The `-type f` option is included so that `find` searches only for regular files and not directories or other file types. The `-exec` option executes the command given for any file found that matches all criteria. In this case, files are compressed with the `gzip` command.

The `CheckExitStatus` function is called after the `find` and passed two strings: one is printed if the command has failed, and the other is a string that indicates whether a failure should cause the script to exit. If `YES` is passed as the second argument, a failure will cause the script to terminate.

The end of the script, which could be considered the main program, uses a `for` loop to iterate over any directory paths passed as command-line arguments:

```
# Main program. Loop over directories passed via command line arguments.
# Compress any large files in each directory tree.
for dirPath
do
    CompressFilesInDirTree "$dirPath"
done
```

For each iteration, the `CompressFilesInDirTree` function is called, passing the directory path as the only argument. The argument is given in quotation marks in case the path includes any spaces, which would cause it to be passed as multiple arguments.

Summary

The Bash Shell plays a vital part in Mac OS X operations, being used at system startup, and for system maintenance. You can use Bash to access a wealth of different Unix commands and combine them to undertake complex operations. These operations would be much more difficult to do with Finder.

In this chapter you learned

- How to use the Terminal application, and the Nano editor, to use Bash interactively and write shell scripts
- The most important Bash commands, for performing fundamental operations on files and text
- About other less fundamental commands, some of which are found only on Mac OS X
- How you write shell scripts with the programming constructs available in Bash, including conditional branching, looping, and functions

In the next chapter, you learn how to work with AppleScript and AppleScript Studio. AppleScript is quite a different beast to Bash in that it is generally used for scripting applications with a GUI rather than Unix commands. Before proceeding, however, try the exercises that follow to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

Exercises

1. With the Terminal application, use Bash interactively to create a compressed archive of all the files of a particular type (that is, with a particular extension) located in your home directory or subdirectories.

First locate the files with the `find` command and copy them to a temporary directory. Rather than using the standard `cp` command, which neglects to copy resource forks on versions of Mac OS X prior to 10.4, use the `CpMac` command in the `/Developer/Tools` directory. To learn more about this command, see the `CpMac` man page. Use `ditto` to compress the archive, and `mv` to move it to a backup directory.

Refer to the man pages of the various commands to learn how they are used.

2. Convert the interactive commands you used to complete Exercise 1 into a backup script. You can use the `history` command to see what commands you typed. For more information, see the `history` man page.

Restructure the commands to make the script more readable and robust if necessary, and don't be afraid to introduce shell programming constructs such as conditional branches and functions. Have the script check for errors and send you an email you if one occurs. (*Hint:* See the man page of the `mail` command.)